SANDIA REPORT

SAND2018-2095
Unlimited Release
Printed February 2018

PIMS: Memristor-based Processing-in-Memory-and-Storage

Jeanine Cook

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

SAND2018-2095
Unlimited Release
Printed February 2018

PIMS: Memristor-based Processing-in-Memory-and-Storage

Jeanine Cook
Scalable Architectures
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-9999
jeacook@sandia.gov

Abstract

Continued progress in computing has augmented the quest for higher performance with a new quest for higher energy efficiency. This has led to the re-emergence of Processing-In-Memory (PIM) architectures that offer higher density and performance with some boost in energy efficiency. Past PIM work either integrated a standard CPU with a conventional DRAM to improve the CPU-memory link, or used a bit-level processor with Single Instruction Multiple Data (SIMD) control, but neither matched the energy consumption of the memory to the computation. We originally proposed to develop a new architecture derived from PIM that more effectively addressed energy efficiency for high performance scientific, data analytics, and neuromorphic applications. We also originally planned to implement a von Neumann architecture with arithmetic/logic units (ALUs) that matched the power consumption of an advanced storage array to maximize energy efficiency. Implementing this architecture in storage was our original idea, since by augmenting storage (instead of memory), the system could address both in-memory computation and applications that accessed larger data sets directly from storage, hence Processing-in-Memory-and-Storage (PIMS). However, as our research matured, we discovered several things that changed our original direction, the most important being that a PIM that implements a standard von Neumann-type architecture results in significant energy efficiency improvement, but only about a O(10) performance improvement. In addition to this, the emergence of new memory technologies moved us to proposing a non-von Neumann architecture, called Superstrider, implemented not in storage, but in a new DRAM technology called High Bandwidth Memory (HBM). HBM is a stacked DRAM technology that includes a logic layer where an architecture such as Superstrider could potentially be implemented.

Contents

List of Figures

List of Tables

Chapter 1

Project History

As stated previously, the project began by investigating a traditional von Neumann architecture in memory. Initial experiments done on a functional simulator showed that the performance improvement was approximately 10x for a small suite of benchmarks. Additionally, through measurement on some of our testbed systems using various applications, we discovered that bandwidth limitations to memory are stressed primarily by computations that involve sparse matrix/vector data. This drove us to re-think the architecture and eventually arrive at the Superstrider architecture, which is described in Chapter 3.

Although Dennard Scaling has driven efficiency and performance improvements over the past several decades, like Moore's Law, the end is in sight. The proposed devices to replace traditional CMOS transistors, such as tunneling FETS and ferroelectric transistors, will enable a reduction in V_{dd} to the point where reliability will be negatively impacted and error correction is necessary. Therefore, we initiated some research as part of this project on error correction algorithms. After reviewing several error correction algorithms, we chose to pursue an architecture that used Redundant Residual Number System representation for integer arithmetic operations. This work is presented in Chapter 4).

Implementing an architecture based on RRNS has implications on the memory subsystem. In a traditional architecture, memory addresses are binary. However, in an RRNS architecture, these addresses are generated in RRNS, causing a slowdown in overall performance of 2-3x. We investigated several methods for memory addressing in an RRNS architecture that showed improvements in performance. These are presented in Chapter 5.

Appendix A contains much of our older published work on our initial PIMS and Superstrider architectures, in addition to documents on work that was done on various aspects of the RRNS-based architecture.

1.0.1 Project Legacy

Through our work in integrating the Superstrider architecture and HBM, several vendors showed interest in collaboration on this, including HPE, AMD, Micron, and Samsung. Upon vendor request, we are trying to identify large-scale scientific applications that are important to the DOE complex that might benefit from the Superstrider/HBM architecture.

This project did launch two primary new research directions. The first is at SNL, in low-energy (reversible) computing. This is now being funded by the ASC Beyond Moore Computing effort. The second is at Georgia Tech. We initiated an ASC-funded Academic Alliance for 3 years to continue work on Superstrider and to leverage GA Tech's CRNCH center that aims to acquire non-traditional computing systems that may eventually be productive in the HPC environment. Superstrider performance will be compared to some of these systems and GA Tech will port important proxy apps to these systems to aid in understanding the benefit to SNL mission applications. Finally, this ASC funding will enable the development of a Superstrider prototype using an HBM integrated FPGA board for testing performance of relevant SNL applications.

1.0.2 Project Metrics

Through this project, we produced the following products (in addition to papers contained in the report):

- 1 patent filed for PIMS, another pending (waiting on legal) for SuperStrider
- SuperStrider simulator open source software release
- 8 peer-reviewed conference papers (1 in addition to papers in the report)
  - Agarwal, R.L. Schiek, M.J. Marinella. Compensating for Parasitic Voltage Drops in Resistive Memory Arrays, 2017 International Memory Workshop
- 7 IEEE Computer Invited Articles (listed below)
  1. Debenedictis, Erik, and R. Stanley Williams. Help Wanted: A Modern-Day Turing. Computer 49.10 (2016) 76-79
  2. DeBenedictis,Erik P. Computational Complexity and New Computing Approaches. Computer 49.12(2016):76-79
  3. DeBenedictis,Erik P. It's Time to Redefine Moore's Law Again. Computer 50.2(2017):72-75
  4. DeBenedictis,Erik P. Computer Architecture's Changing Role in Rebooting Computing. Computer 50.4(2017):96-99
  5. DeBenedictis,Erik P., Jesse K.Mee, and Michael P.Frank. The Opportunities and Controversies of Reversible Computing. Computer 50.6 (2017): 76-80
  6. DeBenedictis,Erik P.,et al. Sustaining Moore?s Law with 3D Chips. Computer 50.8(2017):69-73
  7. DeBenedictis, Erik P.. Computer Design Starts Over. Computer 50.8 (2017): 14-17
- Numerous presentations in various venues

Chapter 2

Impact

This LDRD project has been instrumental in defining a third path in the ASC Beyond Moore Computing effort, which is called non-traditional HPC architectures, that really targets 3D integration of compute and memory. This is now funded work under the ASC Beyond Moore Computing project. In addition, this project established an Academic Alliance with Georgia Tech, which provides funds to continue research in PIMS and other non-traditional HPC architectures. A summary of the impacts of this work is:

1. We have formed active collaborations with several industrial researchers from Micron, HPE, AMD, and Samsung on implementing Superstrider and similar architectures into an HBM technology.
2. We developed and open-sourced a memory-accurate, functional Superstrider simulator that can be used to further develop broader capability in this non-von Neumann architecture. This is being leveraged in academia by GA Tech and in industry by HPE.
3. We quantitatively showed that Superstrider can potentially attain approximately O(1000) performance improvement over the conventionally-implemented algorithm for sparse matrix accumulation. Also showed that performance scales up with tighter coupling and a wider HBM interface (i.e., Superstrider can push the existing limits of HBM bandwidth).
4. We have published 8 refereed papers and have given numerous invited talks at venues including USC ISI, NNSA, and Sandia's Workshop Fault-Tolerant Spaceborne Computing Employing New Technologies.
5. We secured sufficient follow-on funding to continue this work.

Chapter 3

Superstrider

The first paper, *The Superstrider Architecture: Integrating Logic and Memory towards non-von Neumann Computing*, was published in the IEEE International Conference on Rebooting Computing, 2017. This represents our final work and results from this project for the Superstrider architecture. The paper titled, *Superstrider Associatve Array Architecture*, was a Graph Challenge entry and was published in the proceedings of the 2017 IEEE High Performance Extreme Computing conference.

The Superstrider Architecture: Integrating Logic and Memory towards non-von Neumann Computing

Sriseshan Srikanth and Thomas M. Conte
Georgia Institute of Technology
Atlanta, Georgia
Email: seshan@gatech.edu; conte@gatech.edu

Erik P. DeBenedictis and Jeanine Cook
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM
Email: epdeben@sandia.gov; jeacook@sandia,gov

***Abstract*—We present a new non-von Neumann architecture, termed "Superstrider," predicated on no more than current projected improvements in semiconductor components and 3D manufacturing technologies, which should offer orders of magnitude advances in both energy efficiency and performance for many high-utility problem classes. The architecture is described, which is based on computing on row-wide memory words to accelerate sparse matrix algebraic operations that are normally implemented as scalar operations. A cycle-accurate simulation demonstrates potential performance improvements on existing High Bandwidth Memory (HBM) on the order of 50× that increases to 1000× or more when implemented using a fully integrated 3D technology and compared to a simple baseline. Further refinement may change these numbers, but the magnitude of the opportunity suggests further work.**

Keywords—Moore's law; superstrider; processor-in-memory; component; von Neumann; sparse matrix; associative array

I. Introduction

Realization of Moore's law created a world-wide information revolution and economic expansion due to the exponential growth in the number of components per integrated structure [1] (chip) and the computing applications it enabled. Flatlining of line width on chips is evidence that Moore's law is at least changing. Moore's article included a projection of an exponentially increasing number of devices per 2D chip over time, but there are now numerous examples of (memory/storage) chips that are scaling in the third dimension, making it no longer necessary for line width to scale to maintain the vision originally proposed by Moore.

A traditional von Neumann architecture implements the stored-program concept, with both data and instructions being stored in a common memory and moved to the compute unit over a bus. Although the von Neumann architecture and Moore's law are largely responsible for the performance of today's computers, the von Neumann architecture has negatively impacted speed and energy efficiency, particularly with respect to data movement, to a point that renders it unsustainable. The cost of moving data between caches and main memory accounts for a large portion of total energy in several application domains and is cited as the key challenge in future exascale systems [2].

Although memory and storage are now implemented in 3D structures, they are currently being used only in computers of the typical von Neumann architecture. So even if scaling in the third dimension can extend Moore's law, unless we mitigate the von Neumann bottleneck that is created by the bus between processor and memory, we will not see computational efficiency (performance and energy efficiency) grow at the rates that drove economic growth over the past several decades.

In this work, we make use of additional design flexibility brought out by the shift from 2D to 3D [3] and the associated possibility of collocating logic and memory. Some very effective algorithms, such as merging (a type of sorting), become inefficient at large scale simply due to the large amount of data movement between logic and memory chips. Using a merge network as an example, Fig. 1 shows how 3D integration makes the needed interconnection pathways possible and efficient.

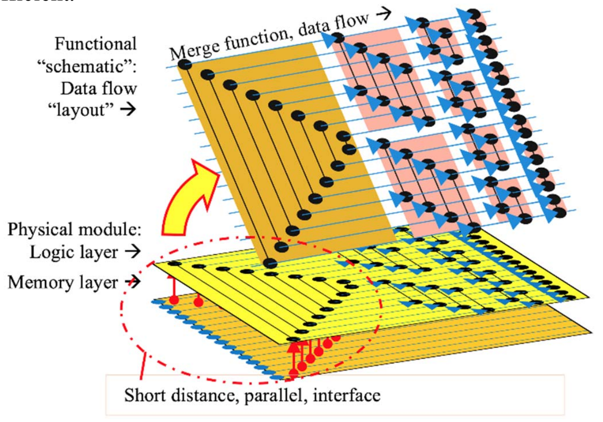

Functional "schematic": Data flow "layout" →
Merge function, data flow →
Physical module: Logic layer →
Memory layer →
Short distance, parallel, interface

Fig. 1: For highest performance, functions are defined as schematics and laid out to reduce communications latency and energy, such as the bitonic merge network on the top. (lower) A 3D physical module with tight coupling between logic and memory allows short connections instead of conversion of signals to high energy levels and off-chip delays. An external von Neumann processor can be interfaced with one of the layers to establish compatibility with existing software. The red curve shows a representative data movement step in merging, which has no off-chip links.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Superstrider belongs to a special class of accelerators, which, in conjunction with a traditional processor, alleviates the von Neumann bottleneck by collocating some computational functions and their associated memory access, enabling increased performance and energy efficiency over a broad class of applications (e.g., data analytics, graph processing, scientific/linear algebra). Superstrider itself is not a von Neumann architecture, as it combines computation and memory into a single component that is directed by a control unit and can be integrated with a standard processor.

II. The Superstrider Architecture

The Superstrider architecture performs a limited set of operations using conjoined logic and memory, enabling it to efficiently execute several algorithms related to sparse matrices, which are the primary computational kernels in several scientific applications. Its organization is non-von Neumann in that it computes very close to memory rows with no processor in the traditional sense. This eliminates the need to move data (referred to as *records* or key-value pairs) from the processor to memory over a bus, thereby eliminating the von Neumann bottleneck.

A. Overview of Superstrider Operation

The operational primitive of Superstrider is to perform in-situ sorting (Section IIB) and compression (combining values of records with identical keys, discussed in Section IIF) at the granularity of a memory row. To achieve this, we organize memory into a tree where the nodes are entire physical rows of the memory, where each row comprises records in sorted order and a *pivot* key that distinguishes subtrees. By setting the fundamental unit of data to be a memory row, we achieve high bandwidth utilization and eliminate bank conflicts.

Superstrider currently implements a fixed set of fundamental algorithms to support sparse matrix computation. In this paper, we focus specifically on the addition or accumulation phase of sparse matrix multiplication. In dense multiplication of matrices, $\mathbf{C} = \mathbf{AB}$, each record c_{ij} of $\mathbf{C}$ is the vector dot product of a row of $\mathbf{A}$ and a column of $\mathbf{B}$, such that $c_{ij} = \sum_k a_{ik}b_{kj}$. If we define $c_{ij}^{(k)} = a_{ik}b_{kj}$, many $c_{ij}^{(k)}$ do not exist when $\mathbf{A}$ and $\mathbf{B}$ are sparse. When the sparsity pattern is too irregular to exploit, this can be treated as a data processing problem on a series of records of the form $\{ i, j, c_{ij}^{(k)} \}$. Superstrider takes vectors of records in the form $\{ i, j, c_{ij}^{(k)} \}$, *sorts* them into a standard form, then *sums* all the values $c_{ij}^{(k)}$ for a given (i, j). This is precisely the accumulation phase of sparse matrix multiply.

Therefore, in the context of such an accumulation phase, an addition operation is used to compress records with identical keys. However, Superstrider implementations of other data processing problems may employ a different collision function [4] to achieve compression. In this paper, we limit the collision function to addition.

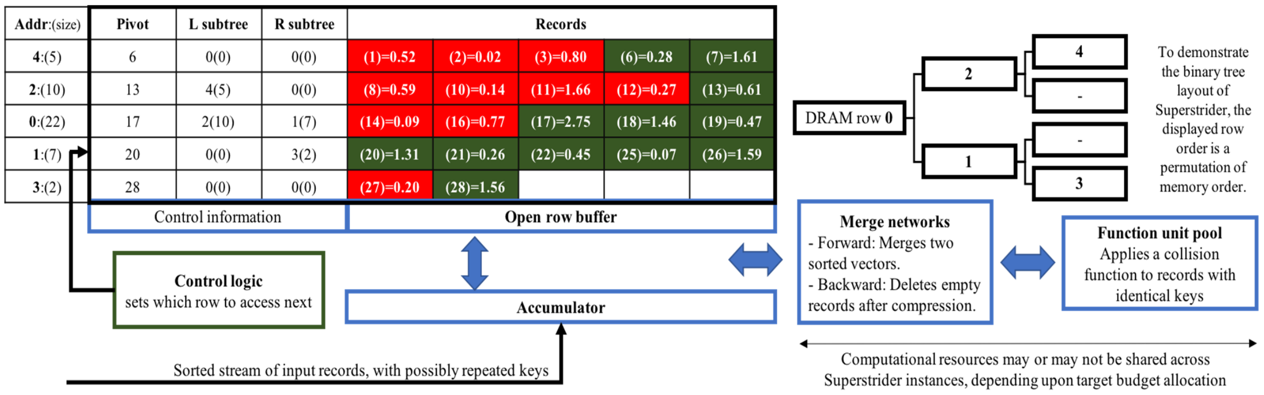
Fig. 2 depicts an overview of Superstrider's architecture and operation. Our Superstrider implementation involves several components:

- A memory array where the rows organized as a tree; where each row stores at most K records.
- An "open row" buffer and an accumulator of size K each, to buffer input and intermediate processed data.
- A merge network to sort records in the open row buffer and the accumulator taken together. When laid out in a different manner and used in reverse, such a network is used to eliminate empty records that manifest as a result of compression.
- A pool of function units consisting of comparators and adders in order to aid with compression.

We dedicate the remainder of this section to explain each of these in further detail.

B. The Merge Operation and Sorted Invariant

Superstrider's sort/merge capability is key to its efficient operation. Shown at the top of Fig. 1, it is based on a bitonic merge network that was first proposed decades ago for use in specialized chips [5]. This network takes two sequences of 8 numbers, each sequence in sorted order, i. e. a bitonic sequence, as input on the left, which flow through the network and are output as single sequence of 16 numbers in sorted order at the right. The vertical lines are comparators, and when two numbers reach the ends of a vertical line, they are compared and swapped so that the larger number is on top. A four-stage network is shown in Fig. 1, where each stage of the network outputs sorted sequences of decreasing size (e.g., bitonic sequences of 8 are merged in the first stage, the second stage merges sequences of 4, the 3rd stage merges bitonic sequences of 2, etc.).



Addr:(size)	**Pivot**	**L subtree**	**R subtree**	**Records**				
4:(5)	6	0(0)	0(0)	(1)=0.52	(2)=0.02	(3)=0.80	(6)=0.28	(7)=1.61
2:(10)	13	4(5)	0(0)	(8)=0.59	(10)=0.14	(11)=1.66	(12)=0.27	(13)=0.61
0:(22)	17	2(10)	1(7)	(14)=0.09	(16)=0.77	(17)=2.75	(18)=1.46	(19)=0.47
1:(7)	20	0(0)	3(2)	(20)=1.31	(21)=0.26	(22)=0.45	(25)=0.07	(26)=1.59
3:(2)	28	0(0)	0(0)	(27)=0.20	(28)=1.56			

Control information
Open row buffer
Control logic
sets which row to access next
Accumulator
Sorted stream of input records, with possibly repeated keys
DRAM row 0
2
1
4
-
-
3
To demonstrate the binary tree layout of Superstrider, the displayed row order is a permutation of memory order.
Merge networks
- Forward: Merges two sorted vectors.
- Backward: Deletes empty records after compression.
Function unit pool
Applies a collision function to records with identical keys
Computational resources may or may not be shared across Superstrider instances, depending upon target budget allocation

Fig. 2: Superstrider architecture and functionality. In a typical operation, such as Addvec (Section IIE), pre-sorted records in the open row buffer and the accumulator are merged using the merge network (Section IIB) and records with identical keys are summed (Section IIF) using the function unit pool. The result is a vector of sorted records, each of which has a unique key. These are then written into the binary tree layout of memory, based on pivot values.

We are able to realize efficient sorting by using only the merge capability of a bitonic network because we enforce the invariant the records in memory rows and the accumulator are always in sorted order. Merging requires just $\log_2(n)$ stages, where n is the number of records being merged. Note that realizing such an efficient sorting paradigm not only aids in insertion/lookup of records into a Superstrider tree, but also aids faster compression.

C. *Control Logic*

The control logic in Superstrider was designed to complement its other unique features, but the control logic will be described only briefly due to space limitations. Superstrider is a state machine overall with one state transition per memory access, with the control logic holding the main part of the state. The control logic receives signals from the data path containing information like the number of records whose key is less than the pivot or whether a subtree exists or not. The control logic produces simple commands like whether to swap the accumulator and open row buffer or to leave them as they are, compute the address or the next row to be accessed, and which function to "call" on the next row.

The most important high-level Superstrider functions include *Addvec* (add vector of records to the tree) and *Normalize*. These operate on a memory row and both use the merge network and the pool of function units to accomplish their tasks. Before describing these functions, we first describe the Superstrider data layout and control fields.

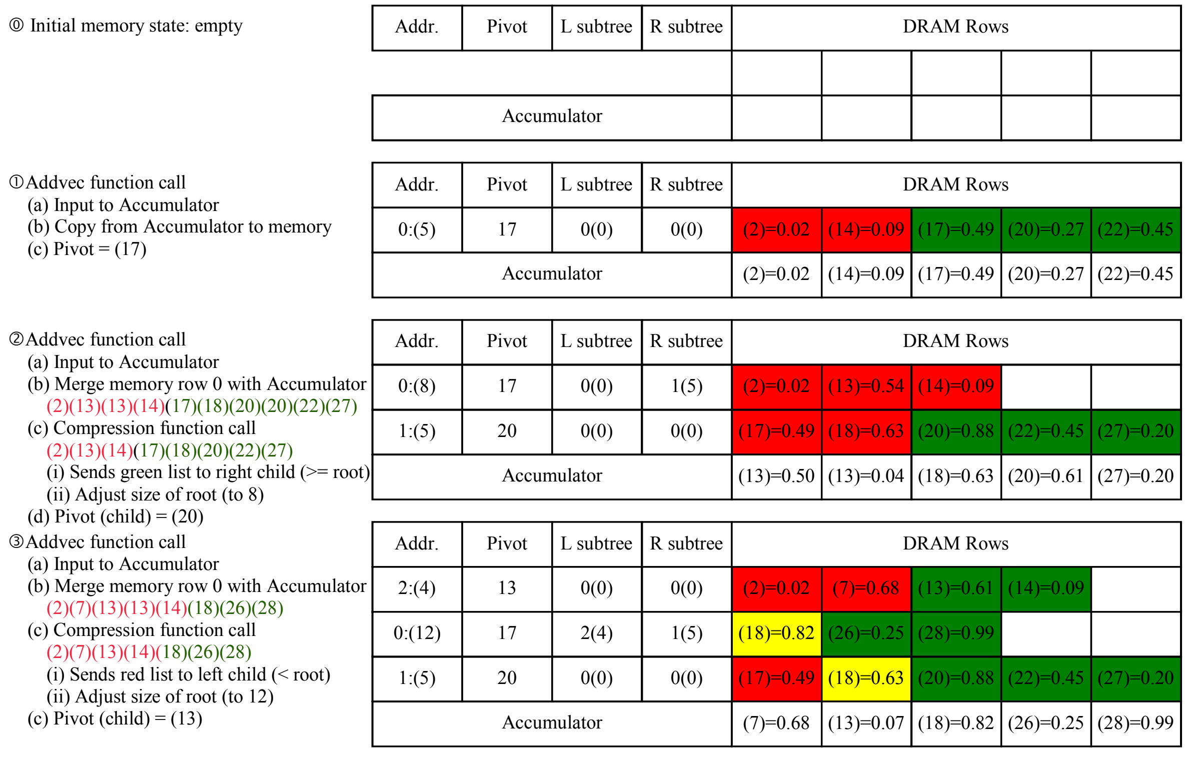
D. *Memory/Data Layout*

A Superstrider row may span several physical memory banks, but for simplicity, we use a single bank to illustrate the data layout. Fig. 2 shows rows of K = 5 records from a Superstrider bank. The fields shown to the left of the red and green records are control fields that are described in Table 1. The contents of each record are in the format "(key) = value", so a term of $c_{ij}^{(k)}$ appears with the notation $(n(i, j)) = c_{ij}^{(k)}$, where $n(i, j)$ is an invertible function mapping two integers to one. While we perform compression (Section IIF) to remove duplicate keys from the same row before adding them to the tree, we explain in Section IIE that it is possible for the same key to appear in different rows. At a later stage, we remove such duplicate keys via normalization (Section IIH).

A bank has an open row, or a row whose contents have been transferred to the open row buffer shown in blue at the bottom of the bank. The control fields to the left of the buffer are likely to be a few dozen bits, as shown in Table 1, with the

Table 1: Control fields

Addr:(size)	The Superstrider row number (Addr) and computed size of the subtree rooted at this row.
Pivot	The pivot value for sorting; is set to $p = K/2$. Elements of the left/right subtree are less than/greater than or equal to this value.
L subtree(size)	The Superstrider row number (Addr) of the left subtree and size of the left subtree.
R subtree(size)	The Superstrider row number (Addr) of the right subtree and size of the right subtree.



⓪ Initial memory state: empty

Addr.	Pivot	L subtree	R subtree	DRAM Rows				
Accumulator								

①Addvec function call
(a) Input to Accumulator
(b) Copy from Accumulator to memory
(c) Pivot = (17)

Addr.	Pivot	L subtree	R subtree	DRAM Rows				
0:(5)	17	0(0)	0(0)	🟥 (2)=0.02	🟥 (14)=0.09	🟩 (17)=0.49	🟩 (20)=0.27	🟩 (22)=0.45
Accumulator				(2)=0.02	(14)=0.09	(17)=0.49	(20)=0.27	(22)=0.45

②Addvec function call
(a) Input to Accumulator
(b) Merge memory row 0 with Accumulator
(2)(13)(13)(14)(17)(18)(20)(20)(22)(27)
(c) Compression function call
(2)(13)(14)(17)(18)(20)(22)(27)
(i) Sends green list to right child (>= root)
(ii) Adjust size of root (to 8)
(d) Pivot (child) = (20)

Addr.	Pivot	L subtree	R subtree	DRAM Rows				
0:(8)	17	0(0)	1(5)	🟥 (2)=0.02	🟥 (13)=0.54	🟥 (14)=0.09		
1:(5)	20	0(0)	0(0)	🟥 (17)=0.49	🟥 (18)=0.63	🟩 (20)=0.88	🟩 (22)=0.45	🟩 (27)=0.20
Accumulator				(13)=0.50	(13)=0.04	(18)=0.63	(20)=0.61	(27)=0.20

③Addvec function call
(a) Input to Accumulator
(b) Merge memory row 0 with Accumulator
(2)(7)(13)(13)(14)(18)(26)(28)
(c) Compression function call
(2)(7)(13)(14)(18)(26)(28)
(i) Sends red list to left child (< root)
(ii) Adjust size of root (to 12)
(c) Pivot (child) = (13)

Addr.	Pivot	L subtree	R subtree	DRAM Rows				
2:(4)	13	0(0)	0(0)	🟥 (2)=0.02	🟥 (7)=0.68	🟩 (13)=0.61	🟩 (14)=0.09	
0:(12)	17	2(4)	1(5)	🟨 (18)=0.82	🟩 (26)=0.25	🟩 (28)=0.99		
1:(5)	20	0(0)	0(0)	🟥 (17)=0.49	🟨 (18)=0.63	🟩 (20)=0.88	🟩 (22)=0.45	🟩 (27)=0.20
Accumulator				(7)=0.68	(13)=0.07	(18)=0.82	(26)=0.25	(28)=0.99

NOTE: (18) does not match ground truth because it has not yet been merged, so is colored yellow.

Fig. 3: Addition of three records to an initially empty Superstrider

remainder of the row divided into K data records. The magnitude of K will vary depending on both the row length and size of user-defined records, but a representative range could be from K=200-2,000. This layout may look similar to a hashed array tree with K-element leaves, however, it is a binary tree with K-elements nodes because of its pivot based organization.

As mentioned previously, Superstrider's primary primitive is merge, with data organized optimally in memory via a tree-based sort. Each row is considered a node in the tree, with the root of Superstrider's overall tree at address zero. We use a pivot in each row that is slightly different than that found in the algorithmic literature for sorting. All keys in the left subtree are less than the pivot; all keys in the right subtree are greater than or equal to the pivot. However, each row has records whose keys can be less than, equal to, or greater than the pivot. In Fig. 2, records in the row whose key is less than the pivot are colored red, the others green, and the records are always sorted.

E. Addvec

Addvec adds a vector with up to K records to the tree present in memory. All the records are added in parallel.

Fig. 3 shows how record vectors of size $K = 5$ are added to an empty memory using a tree-based algorithm. Records whose key is less than the pivot are colored red; all others are shown in green.

The memory is initially empty, which is step 0. In step 1, Addvec copies the input from the accumulator to memory as the root of the tree (i.e., 0:(5)) and sets the pivot to the key of the middle record, which in the example is 17. Pivots do not change once set.

Step 2 shows a new sorted vector of records (length K) in the accumulator being added to the root of the tree in memory. The records in the accumulator and memory are merged, which results in records with identical keys ((13) and (20) both occur twice in the merged record). This merged vector of records (of length $2K$) is then compressed, which adds the values in records with identical keys (e.g., records (13)=0.50 and (13)=0.04 become a single record (13)=0.54). So after compression, we have three records that are less than the pivot value (shown in red, (2), (13), (14)) and five that are greater than or equal to the pivot value (shown in green, (17), (18), (20), (22), (27)). At this point, the algorithm determines which of these vectors to store in memory and which to keep in the open row buffer, and on which child to recursively call Addvec. To maintain consistency of the tree, the Addvec algorithm recursively operates on vectors that are either all less than or all greater than or equal to the pivot value (i.e., all red or all green). Using the pivot (17), a right subtree child is created and written to memory, leaving the keys that are less than 17 in the root (open row buffer) and those greater than or equal to 17 in the right subtree. The size of the root is changed (from 5 to 8) to reflect the new number of records tracked from the root and the child pivot is set to the key of the middle (third) record, or 20.

Addvec is recursively called on the left or right child in the open row buffer, depending upon the relative number of red and green records. This iteration repeats until the proper subtree does not exist, i. e. a leaf of the tree has been reached and the iteration attempts to move below the leaf. At this point, a new leaf node is created with the accumulator value and Addvec completes.

F. Compression

Compression occurs when the same key appears in several records. The merge at the beginning of each step results in equivalent keys being adjacent in the row of records. The pool of function units identifies these adjacent records and with the help of the backward merge network, replaces them with a single record with the common key and a combined value. This shortens the list by some number of records. For accumulation in our sparse vector multiplication example, values are combined by floating point addition, but other applications may use a different collision function.

The simulator that constructed the diagram in Fig. 3 also computes a ground truth representation of the correct **C** matrix. Cells are colored yellow if their value does not match the ground truth. At some point later in algorithm processing (normalization), these two records pictured in yellow will appear in the same row and will be compressed.

G. Multiset, Set, and Standard Forms.

The sort algorithm implemented in memory allows the tree to have an unusual structure, leading to multiple forms. A row may contain records whose key is less than the pivot, but these records can also be in the left subtree. Likewise, a row may contain records whose key is greater than or equal to the pivot, but such records can also be in the right subtree. For multiset form in general, a key can be found in a leaf node and/or in any ancestor node up to the root, or in any combination of these nodes. This property is essential for computational efficiency.

During normalization, all duplicate keys are removed from the tree, converting it from multiset to set form. Allowing only unique keys in the set (i. e., no duplicates) abides by the strict definition of set and requires that all keys in the left/right subtree are less than/greater than or equal to all keys in the row above (parent node) rather than just the pivot.

We define standard form as the set form with the additional constraint that all rows except the last have exactly K records. Standard form is unique.

H. Normalize

Normalization compacts and reorganizes multiset trees with incompletely compressed (yellow) cells and rows of irregular length to the set or standard form. It is recursively called to adjust the division of records between each of its subtrees and the open row, with the division chosen to make the number of records in the left subtree a multiple of K and to fill the memory row completely if there are enough records to do so. Normalization also ensures that all the records in the left/right subtree are less than/greater-than-or-equal to any record in the tree node.

This shifting of records to/from the open row from/to a subtree uses the Normalize function (as well as min and max functions) and the Addvec function, respectively. Addvec shifts records from a row to a subtree. The records are removed from the open row starting at the pivot through the rightmost record and are sent to the subtree, with the pivot being the leftmost record in the new subtree. Normalize moves records from a subtree to the open memory row, using either the min(n) or max(n) functions. These functions are called to find and remove the n smallest or n largest keys in a subtree and return their records. The records are then added to the open row. Obviously, this activity cannot remove more records from a subtree than exist and cannot fill the row beyond its K-record capacity.

If the input tree is in set form, the output of Normalize will be in standard form. All left subtrees will have a multiple of K records, and hence will have rows filled to exactly K records. Since every subtree except the rightmost leaf node of the entire tree is a left subtree of some node, only this rightmost node is not guaranteed to have K records. This corresponds to standard form. If the input tree is in multiset form, the output of Normalize will be in set form. The Normalize function makes use of an estimate of the size of subtrees to compute how to divide the records. However, the number of records in a subtree will change during normalization when records merge. Even though merging may leave rows incompletely filled, Normalize still puts the tree in set form with no duplicates.

III. Experimental Framework

We implemented a memory cycle-accurate simulator for the Superstrider architecture and compared its performance to a von Neumann architecture baseline. We call the simulator memory cycle-accurate because it faithfully preserves cycle timing of memory. However, HBM has a protocol for moving data from the physical memory to the controller. We model this protocol using published timing figures [6], but we extrapolate the protocol to hypothetical HBM successors with wider interfaces where the timing is speculative. At the widest possible interface width, the timing simulated represents a fully integrated logic in memory model.

The simulator accurately counts the number of cycles for the merge network and the function unit pool, however, we do not consider wire delay in these networks. We find that the incremental benefits (even with optimistic performance projections obtained by ignoring wire delays) due to relatively larger networks are rather modest, thereby rendering their complexity and overheads unjustified.

A. *Front End*

A front end generates sorted records to feed the Superstrider HBM+logic structure as well as to measure performance of a von Neumann baseline. The baseline implements no reordering optimizations on the sparse input matrices and does not simulate caches because there is negligible spatial locality in sparse matrix inputs. The baseline traverses the HBM banks as rectilinear memory, i.e., without the binary tree format of the Superstrider algorithm. We use University of Florida matrices [7] as well as pseudo-randomly generated sparse matrices as input to the Superstrider simulator. The latter is especially useful in performance analysis because they enable arbitrarily large inputs, and we use these to demonstrate the benefits of the Superstrider paradigm in Section IV, with the random sparse matrix multiplication input generating 27 million non-zero records as the Superstrider input stream, where the key of each record is a pseudo-random number between 0 and 27 million.

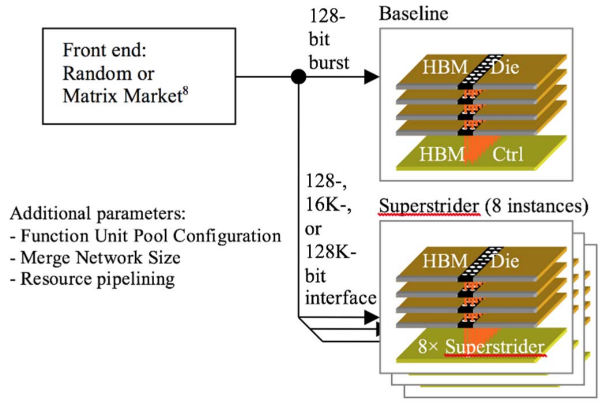


Fig. 4: Experimental framework. Baseline is an HBM stack with controller (8 channels) but no cache anywhere. For compatibility, Superstrider is implemented as 8 instances, one per HBM channel, with a sweep performed over additional parameters.

B. *Memory Model*

To make reasonable comparisons with a conventional processor, the simulator illustrated in Fig. 4 models 8 identical instances of Superstrider, each connected to an HBM channel. Each HBM channel, simulated with 1 rank, comprises 8 16,384-bit wide physical DRAM banks.

Each simulated Superstrider record contains an integer key and a single precision floating point value, or 8 bytes per record. To achieve maximum algorithmic efficiency, each Superstrider row is set as wide as possible, i. e., $K = 2{,}048$ records or 128K bits wide, which is the total row width offered by combining all 8 banks together. A Superstrider row is strided across its 8 banks to avoid bank conflicts, thereby realizing high channel bandwidth utilization.

The physical configuration of a bank as well as the timing parameters that govern row access time are obtained from the High Bandwidth Memory (HBM) JEDEC standard [6]. However, for simplicity, we ignore the overheads due to DRAM refresh and read-to-write delays. We assume that the HBM is clocked at one-fourth the frequency of that of logic.

C. *Simulated Parameters*

In the simulator, we vary several parameters to explore the design space of the Superstrider architecture. We perform experiments to understand the sensitivity to performance of varying the following characteristics: (1) function unit pool configuration, (2) merge network size, (3) "tightness" of the logic/memory integration (interface width), and (4) resource pipelining.

1) *Function Unit Pool Configuration*

A pool of function units are made available to facilitate compression. These can be global or shared across the HBM channels (Superstrider instances), or can be distributed/partitioned or private per HBM channel depending upon the target design budget allocated.

We evaluate three scheduling schemes:

Partitioned/N. The pool of N function units is statically partitioned and distributed across all channels equally.

FCFS Greedy/N. The pool of *N* function units is globally shared across channels and allocation/scheduling is done on a first-come first-serve basis. It is greedy in that the scheduler allocates any available units to an incoming compress request, even if a sufficient number of them is not available to perform the addition in a single time step.

Infinite#. We also evaluate an upper limit policy where there are an infinite number of function units available, guaranteeing constant (single cycle) access time.

2) Merge Network Size

A merge network is responsible for merging the open row buffer and the accumulator and for deleting empty records after compression. We assume that each channel has a merge network associated with it close to the open row buffer and accumulator to minimize wire length. As explained in Section II, this network is a $\log_2 n$-level bitonic structure, with each level taking multiple pipelined cycles depending upon the number of comparators available. For simplicity, we assume that there are enough ports to the network to feed all its comparators simultaneously.

Recall that a Superstrider row spans 8 banks, each of which is 16,384 bits wide. This means that the open row buffer as well as the accumulator can house 2,048 records each. We simulate three merge network sizes by varying the number of single-cycle two-record comparators available: 4, 256, 2,048. However, because we observe low marginal utility from increasing the size of the merge network all the way to support 2,048 comparisons per cycle, we conclude such complexity as unwarranted, and omit presentation of their results.

3) Interface width: Near-Memory vs. In-Memory Logic:

We simulate a near-memory compute paradigm by modeling Superstrider as an HBM controller chip. Although the row buffer is 16,384 bits wide in an example HBM configuration, data is provided to the base layer I/O in bursts that are only 128 bits wide. This means that such a near-memory logic configuration is limited by this narrow burst width, although a memory access reads an entire row into the memory row buffer.

By simulating higher interface widths we increase the "nearness" of near-memory compute, thereby increasing the "tightness" of the coupling between logic and memory, making it in-memory compute at the limit. While a production HBM has a 128 bit interface (burst) width, we simulate interface widths of 16,384 bits and 128K bits using HBM timing. However, in the absence of off-the-shelf implementations, we hypothesize the physical realization and timing of the larger interface widths.

4) Resource Pipelining

As the Superstrider instances are mutually independent, we allow for interleaving between components across these instances, and, thereby benefit from channel-level parallelism.

In addition, we optionally allow for pipelined execution within each Superstrider instance:

Non-pipelined. There is no pipelining between the operation of the components (open row buffer, accumulator, merge network, function unit pool), as they process any given row.

Pipelined. This builds upon the simplistic approach above by allowing adjacent components to overlap execution. For example, as a row is being read out from memory in bursts, it can proceed to the first stage of the merge network in same-sized bursts without having to wait for the entire row to be first read. Similarly, the last stage of the merge network can be overlapped with the first stage of the function unit execution, the last stage of which can be overlapped with the first stage of the deleting network. This fine-grained pipelining can be implemented using FIFOs.

Pipelined with Write Buffer. The pipelining described above is limited to a single row because we need to have finished processing a row in its entirety before we know the address of the next row. As such, there is a window of time where the memory channel is inactive while it waits for the row processing to finish. Subsequently, there is another window of time where the processing logic is inactive when the just-processed row is being written back to memory. To increase the overlap between logic and memory components, we employ a write buffer to store processed rows and flush them out while a subsequent row is being processed.

IV. RESULTS

A. Data transfers

The principal advantage of Superstrider is that it mitigates the von Neumann bottleneck by reducing the number of bandwidth-limiting and energy-consuming transfers between the processor and memory. In conventional processors, cache-line utilization (including hardware prefetching) for sparse matrices is extremely low. In contrast, Superstrider makes effective use of an entire row and there is no extraneous traffic. In fact, to benefit from bank level parallelism and extract maximal algorithmic efficiency, recall that we stride a Superstrider row across its 8 banks, thereby making effective use of 8×2048 byte wide rows at a time.

For an estimation of energy saved due to reduced memory traffic, we count the number of times logic accesses memory at a DRAM row granularity. We find that Superstrider accesses over 121× fewer physical rows from memory than the von Neumann baseline.

We will see in the next section that the amount of computational resources required for orders of magnitude speedup is relatively low. A detailed power model is beyond the scope of this paper, but it is well known [2] that data transfers are the primary contributors to energy consumption. Clearly, the significant reduction in memory traffic described above renders a proportional reduction in system energy.

B. Performance

The Superstrider algorithm reduces the number of transfers between logic and memory thereby saving energy and reducing wasted bandwidth, or in other words, the Superstrider architecture circumvents the von Neumann bottleneck. As we shall describe below, *even the most resource-constrained configuration results in close to 50× performance improvement over von Neumann baseline.* Upon subsequently removing various resource bottlenecks from our Superstrider implementation, simulation shows an additional speedup of close to 80×.

For the memory+logic configurations of Section III, we now present sensitivity of performance of each configuration to the simulation parameters as outlined in Section III C.

Function unit pool configuration. Partitioned/8 yields a speedup of 49-96× for an HBM-based near-memory logic configuration with 4 comparators per channel, depending upon the Superstrider pipelining configuration employed. In other words, allocating just a single function unit and 4 comparators per Superstrider channel yields significant benefits, as shown in Fig. 5.

Partitioned/64 and FCFS Greedy/8 yield identical benefits. In other words, the designer can make a tradeoff between dedicating 8 function units per Superstrider channel for shorter wires/low scheduling overhead, and, sharing 8 units across all channels for improved resource utilization.

In general, we find that FCFS Greedy/64 approaches the performance of Infinite# when 4 comparators are used, meaning that 64 function units are more than sufficient as the bottlenecks are in the sorting network and memory access width.

Merge network size. For an HBM-based near-memory logic configuration, increasing the number of comparators per merge network from 4 to 256 yields an additional improvement of 1.8-2.6×, depending upon the resource pipelining scheme employed. Further increasing the merge network size is not useful as the system is bottlenecked by memory access width.

Interface width. Increasing the interface width from 128 bits to 16,384/128K bits (or by 128×/1024× respectively) renders an additional improvement of slightly less than 2× (for all resource pipelining and function unit pool configurations) when the system is bottlenecked by a mere 4 comparators.

However, upon *also* increasing the number of comparators per merge network to 256, significant additional improvement is seen when the interface width and the capability of the function unit pool are increased, as shown in Fig. 6. The larger merge network is now able to better keep up with data being delivered due to the increased interface width, rendering improved marginal utility of more powerful function unit pool configurations as well.

Further increasing the size of the merge network to support 2,048 comparisons per cycle realizes very modest improvements when increasing interface width to 16,384 bits and larger. Similarly, the fabrication/implementation cost of achieving logic-memory integration all the way to 128K bits is not sufficiently justified by the relative improvement in performance.

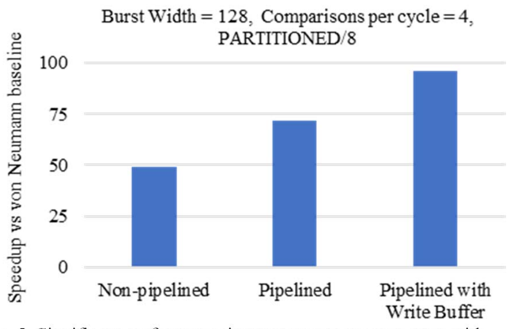


Fig. 5. Significant performance improvements are seen even with simplistic, resource-constrained implementations of Superstrider, owing to alleviation of the von Neumann bottleneck.

Resource pipelining. Improving the degree of overlap between logic and memory access components yields additional benefits (about 2×) in a manner similar to that of improving the function unit pool or merge network's capability, as the system becomes bottlenecked by interface width. This is demonstrated in Fig. 5, but applies to other configurations as well.

In the general scenario, the relative order of efficiency is Non-pipelined < Pipelined < Pipelined with Write Buffer. However, in the scenario where there is little overhead in computation (such as with over 2,048 comparators and 64 function units), using write-buffer based pipelining can be detrimental to performance. This is because without a write buffer, the write-back occurs to the same row, resulting in a single precharge latency incurred upon closing that row, post its write. With a write buffer, however, adjacent reads and writes are to different rows, meaning that there is an additional overhead of row activation and precharge. When there is little overhead in computation, this additional row opening and closing DRAM command latencies are no longer hidden. A row remap memory may be designed to remove this limitation.

V. Comparison with Related Work

Superstrider is a proposed hardware solution to address computational efficiency issues for many algorithms that experience performance degradation due to the von Neumann bottleneck. Sparse matrix multiplication suffers performance loss on current computational platforms due to this bottleneck.

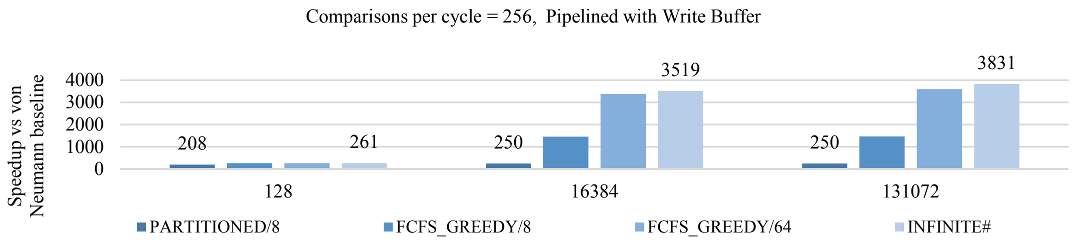


Fig. 6: The utility of increasing the compute resources available to Superstrider is realized only when the memory access bottleneck is loosened. By simulating higher-than-128 interface widths, progressively tighter integration of logic and memory is realized. For space constraints, only the best resource pipelining scheme is shown, although these trends apply to other schemes as well.

Many software solutions have been proposed to address this [8-11]. However, little work has been done to address this issue in the hardware space.

Song et. al. introduce a graph processor architecture that represents graph processing as a sparse matrix algebra problem [12]. They propose a novel node architecture that comprises several modules including memory (cacheless), ALU, systolic merge sorter, matrix reader and writer, control, and interprocessor communication. The systolic merger sorter is used for sorting matrix record indices during matrix operations and is the key to graph processing. The ALU module operates on a stream of sparse matrix elements, making it more efficient than operation of data in a register file as in traditional processor architectures. These graph processing nodes are interconnected in a 3D toroidal configuration to form a 3D parallel processor. Through bit-level simulation models using various graph processing kernels, they show orders of magnitude speedup over commercial systems.

A 3D-stacked logic-in-memory (LiM) system architecture for accelerating graph processing proposed [13] has logic layers stacked between DRAM dies that communicate vertically using through silicon vias. Their customized logic for processing sparse matrix data is integrated with a CAM memory customized to specifically support matrix assembly in the SPGEMM benchmark. Results show over two orders of magnitude of performance and energy efficiency improvement over traditional multithreaded implementations. However, they operate only on compressed data, which means that the input matrices have to be static. Not only do we provide comparable benefits (if not better), our architecture is also capable of supporting inputs that require dynamic insertion. Furthermore, our abstraction provides the potential to implement other data irregular applications by modifying the collision function to something other than addition.

Although we simulated only the accumulation phase of sparse matrix multiply (index sort and merge), this accounts for more than 95% of computational throughput for sparse matrix multiply [12]. Neither of the architectures described above implement as tight an integration of logic and memory, Superstrider implements a unique merge capability, is coupled with HBM rather than a more traditional memory device, and it strides/operates on a "super"-sized, very wide memory word. All of these features together realize very large improvements in computational efficiency.

VI. Conclusion

In this work, we present Superstrider, a 3D architecture that integrates logic in memory to alleviate the von Neumann bottleneck and increase computational efficiency of key scientific algorithms, particularly the sparse matrix accumulation phase in sparse matrix multiplication that suffers from poor cache utilization. We show that Superstrider can potentially provide orders of magnitude speedup for accumulation compared to conventional von Neumann architectures and processing.

This is as a result of improved bandwidth utilization and we attribute this to its unique operational primitives (merge and compress) that operate at the granularity of a memory row, and its novel tree-based representation of sparse matrices. Even the most resource-constrained HBM configuration simulated results in a 50× performance improvement. Furthermore, reasonably increasing the tightness of logic-memory integration and the amount of computational logic resources available renders a *further*, potential improvement of 80×.

VII. Future Work

The authors have already performed additional theoretical work on Superstrider's generality, reported in Ref. 3. The memory "tightness" can be generalized into an incremental development strategy, like Moore's law. Also, the floating point add and multiply operations are a mathematical semi-ring. If the semi-ring is replaced by, for example, addition and minimum, Superstrider can perform graph operations useful in, for example, big data computations. The generalization of Superstrider is an associative array processor.

It should be possible to broaden Superstrider's function beyond "accumulation." For example, store matrices **A** and **B** in Superstrider and create an empty tree for **C**. Then run a function that computes **C** = **AB** with no processor intervention.

Hardware demonstrations should be possible, even without building any hardware. There are companies selling HBM controller IP that offer samples of their product as an FPGA connected to an HBM stack. If these companies would allow augmentation of the controller IP with Superstrider function, perhaps these product samples could become the first production Superstrider hardware.

References

[1] G. E. Moore, "Crammng more components onto integrated ciruits," *Electronics Magazine*, vol. 38, no. 8, 1965.

[2] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," presented at the *Proceedings of IEEE International Symposium on Workload Characterization*, 2013.

[3] Zhao, J., Zou, Q., & Xie, Y. (2017). Overview of 3-D Architecture Design Opportunities and Techniques. IEEE Design & Test, 34(4), 60-68.

[4] J. Kepner, "Spreadsheets, Big Tables, and the Algebra of Associatve Arrays," MAA & AMS Joint Mathematics Meeting, Jan 4-7, 2012

[5] Batcher, K. E. (1968). "Sorting networks and their applications". Proc. AFIPS Spring Joint Computer Conference. pp. 307–31

[6] High bandwidth memory (hbm) dram. [Online]. Available: https://www.jedec.org/standards-documents/results/HBM

[7] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, 2011.

[8] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - a high level linear algebra library for gpus and multi-core cpus," in *International Workshop on GPUs and Scientific Applications*, 2010.

[9] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, "GPU-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, 2015.

[10] Deveci, M., Trott, C., & Rajamanickam, S. (2017, May). Performance-Portable Sparse Matrix-Matrix Multiplication for Many-Core Architectures. In Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International (pp. 693-702). IEEE.

[11] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix-matrix multiplication for the gpu," *ACM Transacations on Mathematical Software (TOMS)*, vol. 41, no. 4, 2015.

[12] W. S. Song, J. Kepner, V. Gleyzer, H. T. Nguyen, and J. I. Kramer, "Novel graph processor architecture," *Lincoln Laboratory Journal*, vol. 20, no. 1, 2013.

[13] Zhu, Qiuling, et al. "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware." *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013

Superstrider Associative Array Architecture

Approved for unlimited unclassified release
SAND2017-7089 C

Erik P. DeBenedictis, Jeanine Cook
Center for Computing Research, Sandia National Labs
P. O. Box 5800 m/s 1319
Albuquerque, NM 87185-1319
epdeben@sandia.gov, jeacook@sandia.gov

Sriseshan Srikanth and Thomas M. Conte
School of Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
seshan@gatech.edu, conte@gatech.edu

***Abstract*—We define the Superstrider architecture and report simulation results that show it could be key to achieving HIVE hardware goals. Superstrider's performance comes from a novel sparse-to-dense stream converter, which relies on 3D manufacturing to tightly couple DRAM to an internal network so operations like merging and parallel prefix can be performed quickly and efficiently. With the ability to use the stream converter as a programming primitive, the memory-bound low-level graph operations that we are aware of speed up substantially. We give special attention to triangle counting in this paper.**

Simulations detailed elsewhere[1] show 50-1,000× improvement in speed and energy efficiency. The low end of the range should be achievable by constructing a custom controller for current High Bandwidth Memory (HBM) where the high end would require fully integrated 3D that is on roadmaps for the future.

Keywords—Superstrider; Moore's law; 3D chips; sorting, processor-in-memory; sparse matrix; backpropagation; associative array; GraphChallenge

I. INTRODUCTION

The world went through an information revolution driven by the exponential growth of microprocessor performance and DRAM size popularly called Moore's law. The flat lining of microprocessor performance has led to talk of "Moore's law ending" and dire consequences to the economy. However, the vertical axis on the graph defining Moore's law in Fig. 1 is clearly labeled "number of components per integrated function" (chip) not "microprocessor performance" and some companies find they can maintain growth in component count by using the third dimension to stack more devices per unit surface area. It seems the economy and Moore's law are healthy, but the top-level division of computers into microprocessors and DRAM must change.

Could GraphChallenge and the associated HIVE hardware project have a role in the future direction the industry? The organizers of GraphChallenge make a compelling case that important new classes of applications conflict with von Neumann's division of computers into processor and memory, which we argued in the paragraph above is limiting the industry as a whole. If the HIVE project develops a graph architecture that scales in a way that is compatible with physical-level semiconductor roadmaps, perhaps HIVE could help define the mainstream direction of the industry?

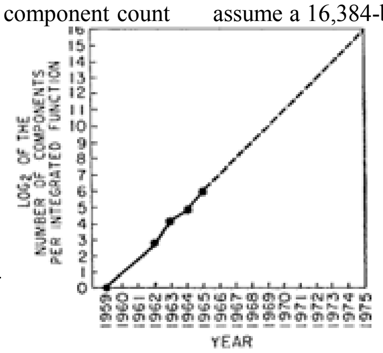


Fig. 1. Graph defining Moore's law from Moore's paper,[2] projecting that device count per chip will rise exponentially for the decade 1965-1975.

A. *Moore's law and 3D*

Even though industry wanted to discover a new transistor for logic, it actually developed new memory devices that can be manufactured in 3D. Developments in 3D chips have been compelling enough that there are now roadmaps for further staged development.[3] The roadmaps show a path through intermediate technologies with progressively more features in the third dimension, a greater variety of devices, and more efficient transfer of information along the third dimension.

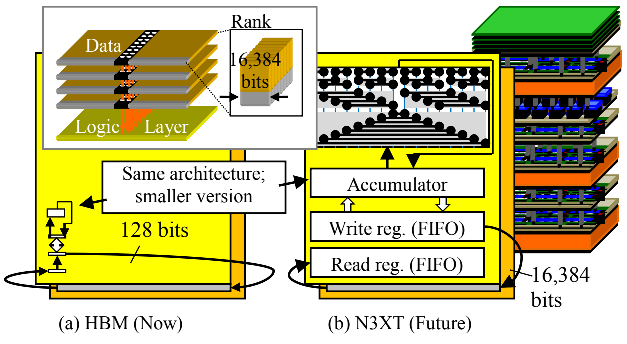
There are two 3D commercial product categories right now. Stackable DRAM is the first, available as High Bandwidth Memory (HBM[4], illustrated in Fig. 2a) and Hybrid Memory Cube (HMC[5]). HBM transfers 16,384-bit DRAM rows as 128 cycles of 128 bits. 3D flash storage is the second product category.

There are several visions for long term development of fully integrated memory, storage, and logic, one example being N3XT[6] illustrated at the top of Fig. 2b. The tighter integration of the proposed N3XT system would have no reason to limit the width of the memory interface, so for consistency we assume a 16,384-bit interface that operates in 1 cycle. In fact, a scale up path can be defined based on the tightness of logic-memory integration, as shown in Fig. 2c.

II. APPROACH

Fig. 3 shows the basis of our approach for exploiting 3D memory to improve graph and other computations. Industry's historic choice to manufacture DRAM and microprocessors as separate chips caused a "data modality" problem. Fig. 3a applies to essentially any multi-chip computer system, showing yellow logic chips, orange memory chips, and connected by gray chip-to-chip interconnect. Algorithms where information

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.



HBM Row: 16,384 bits =

Interface width ×	Cycles	Possible network size
128 bits (HBM)	128	8 = 4 × 2
256	64	24 = 8 × 3
512	32	64 = 16 × 4
1,024	16	160 = 32 × 5
16,384 (N3XT)	1	4,608 = 512 × 9

(c) Scaling scenario

Fig. 2. (a) HBM physical structure on top and Superstrider layout on bottom. (b) N3XT physical structure on top and functionally similar but scaled Superstrider layout on bottom. (c) Hypothetical scaling sequence based on doubling interface with, halving clock periods, and increasing merge network depth.

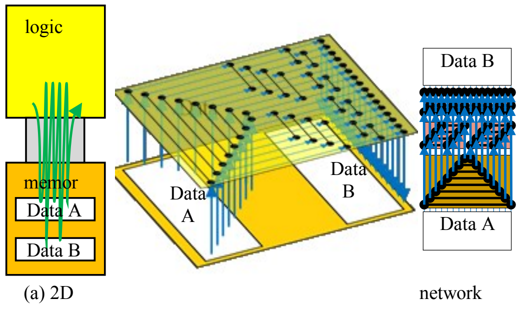


Fig. 3. (a) 2D systems comprise logic and memory chips, with the green curve illustrating a mixed logic-memory calculation that is inefficient precisely because of the partitioning into logic and memory. (b) A 3D system with tight coupling between logic and memory avoids high latency paths, bandwidth bottlenecks, and conversion of signals to high energy levels for off-chip interconnects. The blue curve shows a representative data movement step in sorting. (c) The sorting network used for the 3D example.

dependencies alternate many times between logic and a large memory (i. e. a memory too large for implementation by cache) inevitably involve moving data across the interface between the two twice per repetition, or shifting modes, as illustrated by the green curve. While the efficiency of electronics entirely within a chip improved with Moore's law, the speed and energy efficiency of chip-to-chip interconnect scaled more slowly, creating a bottleneck.

A. *Sorting networks*

We remove the bottleneck using the additional design flexibility offered by 3D. If memory can be stacked or monolithically fabricated adjacent to logic along the chips' planar faces, not only are wires shortened by the tighter packing allowed by the additional dimension, but the modality problem is relieved because data movement between logic and memory no longer needs to traverse chip-to-chip interconnect.

The shift to 3D offers algorithmic benefit as well, which we will discuss using sorting as an exemplary algorithm class. Sorting has been studied extensively for the structures in Fig. 3a and b, where sorting networks, such as the O($\log^2 n$)-step bitonic sort[7] in Fig. 3c, have been known to be faster for decades – but there is more to the story.

Fig. 3c shows the data dependency diagram for one merge stage of bitonic sort. The diagram shows the merging of two sorted lists of 8 data records, which occurs in 4 stages. The first stage does pairwise comparison of all 16 values, swapping the records so the largest one is on the left. The second and later stages do pairwise comparisons as well, but on more groups of fewer records. The pattern can extend to data that fills one or more rows of DRAM. When implemented with the structure in Fig. 3b, the comparisons and swaps can be laid out on the surface of the logic layer as shown. The interface between logic and memory will be via wires crossing the short gap between the large 2D surfaces.

However, the literature for sorting algorithms follows software conventions, which are modeled on a von Neumann computer and hence the structure in Fig. 3a. A von Neumann computer was originally viewed as doing one thing at a time – although recent multicore computers can do up to, say, a dozen things at a time. However, no von Neumann computer comes anywhere close to being able to simultaneously operate on the data in an entire row of DRAM at once. This is the rationale for the Wikipedia page on "Sorting Algorithm" saying "[p]ractical general sorting algorithms are almost always based on an algorithm with average time complexity (and generally worst-case complexity) O($n \log n$)."[8] The best known such algorithm is Quicksort.

Which sorting algorithm is fastest thus depends on whether Fig. 3a or b is the model for implementation. A von Neumann computer does the compare and swap operations one at a time, raising the bitonic sort's complexity from O($\log^2 n$) to O($n \log^2 n$). Thus, Quicksort's O($n \log n$) complexity makes it faster than bitonic sort's O($n \log^2 n$) complexity on a von Neumann computer, but the winner shifts to bitonic sort for implementation in 3D using Fig. 3b, allowing sorting O($\log^2 n$) steps, which is much faster.

B. *Superstrider block diagram*

Superstrider comprises a DRAM memory bank connected to a functionally enhanced butterfly network and a control system, as illustrated in Fig. 4. Functionally, Superstrider is in a never-ending loop reading and immediately writing back DRAM rows. However, the control system selects the row and also uses functional features added to the butterfly network like reduction and parallel prefix to modify the data between read and write back.

Here are a few more details: A small portion of the DRAM row width (a few dozen bits) is dedicated to control fields as indicated in Fig. 4, but the rest comprises *K* data records. There is also an accumulator that, in conjunction with the DRAM's

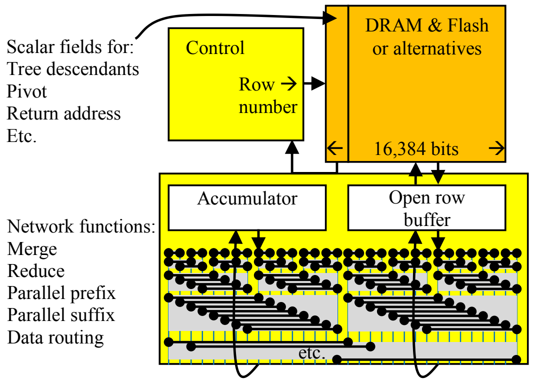


Fig. 4: Superstrider block diagram. A memory bank connects to the logic across its entire width, typically 16,384-bits. The logic comprises a butterfly network with functions listed, connected to both the memory and an accumulator (in the historical sense of the word). A control system implements a basic cycle where DRAM is read and written back continuously as fast as possible. Superstrider function change data between read and write, including storing data in the accumulator.

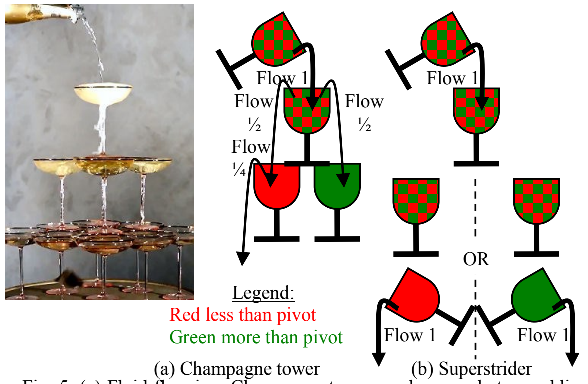


Fig. 5. (a) Fluid flow in a Champagne tower, such as used at a wedding reception, and a tree insertion algorithm. Imagine a Champagne glass holds K red or green records in lieu of Champagne molecules. Pouring K records into a glass or tree node recurses to two similar operations of size $K/2$, ultimately disturbing nearly every glass in the tower. (b) Superstrider can sort based on comparing keys with a pivot, where the comparison result is illustrated by color, yielding asymptotically fewer steps. If the amount poured at one time is the same as the capacity of a glass tree node (K records), sorting the $2K$ items by color is guaranteed to produce at least K items of the same color. Recursion will be needed only down the branch with the predominate color (i. e. tail recursion) and the algorithm will have just $O(\log N)$ steps for K records. (Photo https://vimeo.com/114254175, labeled for noncommercial reuse.)

open row buffer, form a $2K$-record buffer; the butterfly network actually operates on this double-length buffer. While not shown, the memory is accessible to an external, von Neumann-type, processor.

C. *The sparse-to-dense stream converter*

Imagine the pyramidal Champagne tower in Fig. 5a to be a physical analogy of element insertion into a binary tree data structure, where each Champagne glass corresponds to a DRAM row holding K red or green records. A regular tree insertion subroutine searches from the root downward through descendant nodes until the proper one has been found, after which the record is stored in its proper place and the subroutine returns. However, pouring Champagne into the tower has a feature not found in any computer algorithm (as far as we know). Most of the Champagne poured into the top glass does not end up in its final glass after just one pouring, but will temporarily reside in glasses further up the tree until a later pour causes it to move downward.

The problem is resistance to parallelization. If K records are added at the root all at once, like pouring a glass of red and green records into the top glass, there will be more groups of fewer records moving at each level, as shown in Fig. 5a. Since each glass is analogous to a DRAM row, almost all the DRAM rows in the entire system will be accessed for even modest values of K, which is why the structure in Fig. 3a experiences a lot of inefficient data movement between logic and memory chips for many problems.

Fig. 5b illustrates the idea behind Superstrider's sparse-to-dense stream conversion. When K records are poured into a glass that contains another K records, Superstrider sorts these $2K$ records immediately and puts them back in the glasses, but it carefully picks which glass gets the red versus green records. Using the terminology of tree algorithms, each row has a pivot. We'll color records red if their key is less than the pivot and green otherwise. There are now two scenarios corresponding to the bottom halves of Fig. 5b. If sorting reveals K or more red records, one glass will be entirely red and the other glass will have mixed colors. If there are less than K red records, there must be more than K green ones, so the previous statement will be true with the colors interchanged. Now pour the glass where all the records are the same color down just one subtree and leave the other glass in its position in the tree with both colors.

The catch is that the records are not properly added to the subtree that is not visited. So does Superstrider go back and visit the subtree with incompletely processed records? No. That subtree is ignored for the time being. Graph problems typically do long sequences of additions, so subsequent sorts of that tree node will eventually produce enough records of the other color that the subtree will be visited naturally. Simulation shows this to be very efficient for long sequences of additions, yet a cleanup phase, which we call normalization, is required at the end to move laggard records to their proper destination.

The step count reduction is significant and arguably an "order reduction." Adding a single element to a tree is an $O(\log N)$ operation in standard algorithm theory, where N is the number of elements in the tree. Superstrider can add K elements in this amount of time, making the step count per record added $O(\log N)/K$, which captures parallelism but is not an order reduction. However, DRAMs are designed to be refreshed in 8,192 cycles, meaning they actually have 8,192 rows at some low level of the DRAM electronics. If the number of rows is fixed, $K \propto N$, and the step count reduces to $O((\log N)/N)$, which is an order reduction. This argument is rather abstract. If the reader prefers, Ref. 1 shows speedups of 50-1,000×.

D. *Connection to associative arrays*

The preceding description spoke of a 16,384-bit DRAM row being divided into a few dozen control bits and the remainder into K records. In the notation of associative arrays,[9] each record is of the form $\{ k, v_k \}$, where k is a key and v_k is a value. If the record is an int and a float, the record is 64 bits and K is about 256 records. The DRAM may contain multiple arrays such as **A**, **B**, and **C** where $\mathbf{A}[k] = v_k$.

Previous sections of this paper only referred to sorting records, but the butterfly network also uses the network functions listed in Fig. 4 to identify collisions and compress records. This means the sparse-to-dense stream converter performs accumulation part of sparse matrix multiply.

The Superstrider concept is generally compatible with generalizations of associative arrays, but low-level hardware would need to process diverse data types and aggregations beyond what we have actually studied. The Superstrider concept should support associative arrays defined as $\mathbf{A}[\mathbf{k}_i] = v_i$, where **A** is the associative array, $\mathbf{k} = (k^1 \ldots k^d)$ is an array of indices of primitive types, and v may be a data structure of primitive types. Furthermore, even values that are data structures must be an element of a semiring, in which case there will be application-specific operators corresponding to multiplication and addition ($\otimes$ and $\oplus$).

Given the previous definitions, Superstrider could execute graph algorithms, like shortest path, where the values are floats or integers (albeit extended to include ∞) and the operation is $\oplus.\otimes \equiv$ min.+.

While certain simple kernels can be expressed in associative array syntax like $\mathbf{C} = \mathbf{A}$ +.* $\mathbf{B}$, many of the more complex algorithms involve hundreds or thousands of lines of conventional computer code interspersed with calls to compute-intensive associative array operations. Reproducing an example from Ref. 9, the Bellman-Ford algorithm for finding the shortest distance from a node s to all other nodes in a graph **A** appears below, where only the red characters would be performed by Superstrider.

```
Bellman-Ford (A, s)
d = ∞
d(s) = 0
for k = 1 to N-1
    do d = d min.+ A
if d ≠ d min.+ A
    then return "A negative-weight cycle exists"
return d.[9]
```

E. *Superstrider programming strategy*

The sparse-to-dense stream converter becomes a programming primitive of sorts, so we owe the reader a brief lesson on how to program with it. This introduction will lead up to the discussion of triangle finding requested by the Graph Challenge.

Sparse matrix multiply $\mathbf{C} = \mathbf{AB}$ can be performed with multiplications on a host or entirely stand alone. For the first option, Superstrider is initialized with an empty **C** matrix. The host then sends Superstrider vectors of K records of the form $[i, j] = c_{ij}^{(k)}$, where i and j are matrix indices, $c_{ij}^{(k)}$ is a contribution to a matrix element, with superscript $^{(k)}$ distinguishing between the contributions. After sending all the records, the host would command Superstrider to do the normalize function, which cleans up the representation so it could be read out by the host in a packed lexicographic order.

Alternatively, Superstrider could compute $\mathbf{C} = \mathbf{AB}$ autonomously, where matrices **A** and **B** are already stored in Superstrider's memory. This requires first transposing **A** from $\mathbf{A}[i, k] = a_{ik}$ to $\mathbf{A}^t[k, i] = a_{ik}$. This is accomplished by Superstrider accessing **A** internally, interchanging the indices, and sending the records into the sparse-to-dense converter as $\mathbf{A}^t$. This entire process is performed by the hardware in Fig. 4. Reading $\mathbf{A}^t$ later on will produce the records in lexicographic order by what was originally the second index. The actual multiply is then performed by streaming $\mathbf{A}^t$ and $\mathbf{B}[k, j] = b_{kj}$, each in lexicographic order, whereby the products $\mathbf{C}_{ij}^{(t)} = \sum_{k=1,t} a_{ik}b_{kj}$ can be formed by data close together in the streams. The products are then sent to the sparse-to-dense stream converter to creates **C**, using collision of the keys to perform the addition required in sparse matrix multiplication.

The delta rule in backpropagation of neural network learning, is defined as $\mathbf{C} = \mathbf{C} + \mathrm{ab}^t$, which we will perform with no fill-in in two steps. First, $\mathbf{C}[i, j] = c_{ij}$ is streamed along with $a[i] = a_i$ to produce a temporary matrix $\mathbf{T}[j, i] = \{ c_{ij}, a_i \}$. Note that **T**'s indices are reversed, so its records contain the elements of $\mathbf{C}^t$. In addition, **T**'s records are a data structure with both an element of $\mathbf{C}^t$ and an element of a. Secondly, $\mathbf{T}[j, i] = \{ c_{ij}, a_i \}$ is streamed along with $b[j] = b_j$ to produce $\mathbf{C}[i, j] = \mathbf{T}_{ji}.c_{ij} + b_j * \mathbf{T}_{ji}.a_i$, which naturally adds to the existing **C** when sent to the sparse-to-dense converter due to collisions. Since $\mathbf{T}_{ji}$ is a data structure, the notation $\mathbf{T}_{ji}.a_i$ represents the second element of $\{ c_{ij}, a_i \}$ given above (sorry, we need better notation for data structures).

F. *3D Streaming and triangle counting*

Superstrider is essentially a linear algebra machine, so it should be able to execute any triangle-counting algorithm that can be expressed as linear algebra. The authors' attention was directed to the semi-streaming triangle-counting approach of Becchetti in Ref. 10, a paper from 2007. In short, we would put Becchetti's entire algorithm into Superstrider, but this simple statement has more to it than may be immediately apparent.

We propose going a step beyond Becchetti's view of semi-streaming.[10] Becchetti sees a computer as a CPU and DRAM in one unit and "disks" (yes, the article uses the word for rotating storage) on the other side of the cabinet. The large edge-containing matrices are on the disk but DRAM is only deemed big enough to hold vertex-containing matrices. If one were to implement Becchetti's system today, the disk storage would probably be 3D SSD. So when we say we would "put the entire algorithm into Superstrider," we envision 3D chips, such as N3XT in Fig. 2b,[6] with the ability to integrate both RAM-equivalent and storage-equivalent storage right on top of the same logic base chip. This leads to a second interpretation, illustrated in Fig. 4, where Superstrider's memory includes both DRAM and Flash in different ranges of row addresses, yet all appearing to Superstrider as a memory bank. This would allow Superstrider to, for example, multiply a matrix physically stored in Flash by one physically stored in DRAM just by properly specifying the row addresses. Let's call this

"3D steaming" because it works like streaming, but moves data just a few microns in the vertical direction.

Ref. 10's method is based on tagging vertices with random numbers or permutations of node numbers (π(.) in the document). With the possible exception of having a von Neumann host processor generate the permutations π(.), Superstrider could implement the algorithms in Ref. 10's using the linear algebra functions discussed above.

We therefore propose supporting triangle finding with a change in algorithm. Line 15 of figure 5 in Ref. 10 is executed when a triangle has been found, although in Ref. 10 it only triggers a counter increment. We propose that this line additionally create a 3-field record containing the triangle and send it to a sparse-to-dense stream converter designated to hold the output triangles.

We believe the algorithm as outlined so far will function, but it will still be probabilistic, generating the same triangle many times (the duplicates will be removed by the stream converter) and it may take a lot of runs to generate the very last triangle. So we propose modifying the tagging to effectively remove nodes once it has been determined that they have produced all the triangles they will ever produce. This can be done by tagging nodes with ∞ in lieu of π(.).

III. Experiments

We created a cycle accurate Superstrider simulator or 1,500 or 3,500 lines of C++, depending on whether the user interface code is counted. The simulator compares the accumulation phase of sparse matrix multiply against a von Neumann architecture baseline.

A. Memory banks

To make reasonable comparisons with a conventional processor, the simulator illustrated in Fig. 6 models 8 identical instances of Superstrider, each connected to an HBM channel. Each HBM channel comprises 8 16,384-bit wide physical DRAM banks. To consider the extreme range of algorithmic efficiency, each Superstrider row is set as wide as possible, i. e., $K = 2{,}048$ records or 131,072 bits wide.

The physical configuration of a bank and the timing parameters were obtained from the High Bandwidth Memory (HBM) JEDEC standard,[4] however we speculated on the timing of hypothetical HBM successors with wider interfaces per the scale up path in Fig. 2c.

B. Front end

A front end generates vectors of K sorted records to feed both Superstrider and the von Neumann baseline. The front end does not reorder the sparse matrices and the baseline does not simulate caches because there is negligible spatial locality in sparse matrices. The baseline traverses the HBM banks as rectilinear memory, i. e., without the binary tree format of the Superstrider algorithm. The front end supports matrices downloaded from the University of Florida[11] sparse matrix collection as well as pseudo-random sparse matrices generated on the fly. However, the results in this paper are for pseudo-random matrices with 27 million records where the keys are randomly chosen from a space of 27 million possibilities.

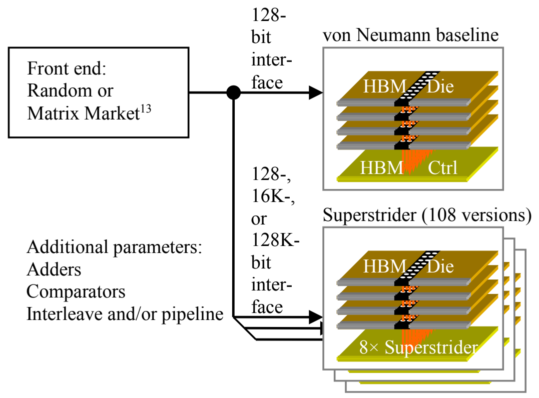


Fig. 6: Experimental framework. Baseline is an HBM stack with controller (8 channels) but no cache anywhere. For compatibility, Superstrider is implemented as 8 instances, one per HBM rank or channel, with a parameter sweep over additional parameters.

C. Simulated parameters

In the simulator, we vary the following five parameters to explore the design space: (1) the width of the logic/memory interface, (2) the size of butterfly network, (3) the number of adders comprising the adder network for collisions, (4) effects of adder partitioning schemes across Superstrider instances, and (5) resource pipelining.

IV. Results

A. Data transfers

The principal advantage of Superstrider is that it mitigates the von Neumann bottleneck by reducing the number transfers between the processor and memory, and the number of internal memory accesses. In conventional processors, cache-line utilization (including hardware prefetching) for sparse matrices is extremely low. In contrast, Superstrider makes effective use of an entire row.

The simulator counts the number of DRAM row accesses to estimate energy saved due to reduced memory traffic. Superstrider accesses 1/121 as many physical memory rows as the von Neumann baseline on the random test case, although this fraction is highly application dependent.

B. Performance

The vertical axis of the 3D bar charts in Fig. 7 is the speedup of Superstrider over the von Neumann baseline. Two messages are clear even at the low-level of detail in this paper.

1. Even the thinnest bars show a speedup of 50× or more, making Superstrider a candidate for implementation as a controller for off-the-shelf HBM memory.

2. Both the 16K and 128K interface widths include speedups of 1,000× or more, although only with 256 or more comparators. In simple terms, Superstrider can make very good use of tight memory-logic integration, but the interface can be

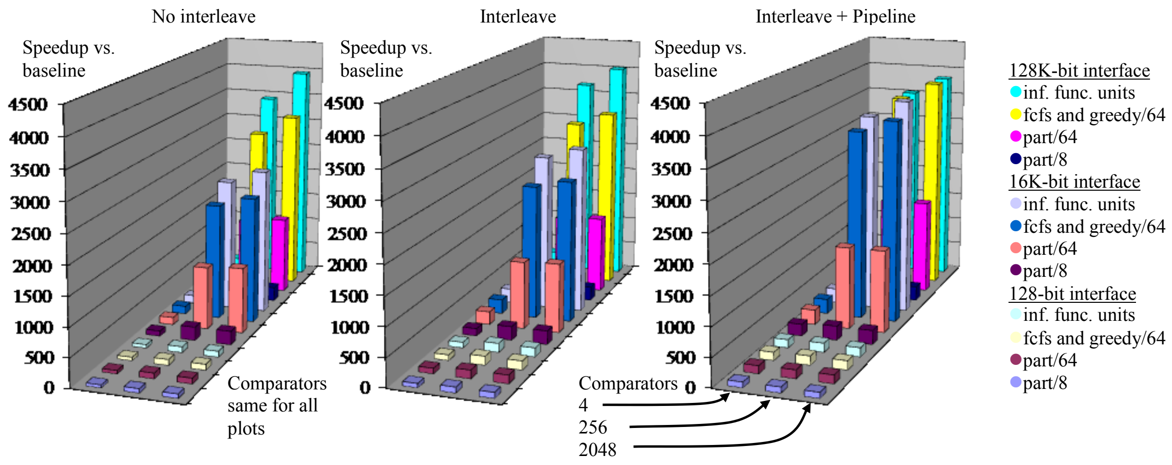


Fig. 7. Superstrider simulation results over the parameter sweeps discussed in the text. The vertical axis represents speedup over a non-cached sequential HBM driven by a microprocessor. The conspicuous result is that performance is high in the back, right hand row. This area corresponds to tight processor memory coupling (wide HBM burst width) and a large number of ports on the merging network. The other parameters are well-known techniques from microprocessor architecture, such as pipelining and interleaving. While these other parameters yield the expected benefit, it appears small in the context of the benefits of the non-von Neumann architectural components.

"choked" by insufficient computational resources. The resource is the comparators in this case, but see Ref. 1 for an explanation of this component and the other parameters.

V. Conclusions

In an environment rife with computer technology efforts to redesign computers from the devices up, we propose to address graph problems through carefully chosen computer architecture and algorithms yet based on just roadmaps of 3D chips. Our result of 50× immediate benefit and 1,000× or more in the long term is good enough to drive the industry and is on par with the more radical approaches.

Instead of writing a specific algorithm or buying or building a specific instance of hardware, we considered a computer whose design is parameterized by the date of manufacture and running software that adapts to the hardware as it changes. The principle parameter is the "tightness" of the logic-memory interface, where roadmaps can be consulted for a rough schedule of this parameter over time. We then created a cycle-accurate simulator and applied it to the most challenging computational kernel, which is sparse matrix multiplication and specifically the accumulation phase.

Algorithm theory showed that the tightening interface between memory and logic will make hardware implementation of sort, merge, parallel prefix, and some other network operations progressively more effective, where the "von Neumann bottleneck" makes them poor choices today. Using the principle of Occam's razor, we packed Superstrider with as much of the new hardware as possible and as few traditional components as absolutely necessary .

A. Loose ends

Superstrider is a chip-scale solution, but we need a way to glue *n* Superstriders together have the same effect as an *n*× bigger one.

We need to turn the simulator into a low-level linear algebra library and then run triangle counting algorithms, as requested by GraphChallenge.

It should be possible to configure Superstrider's internal state machine to do functions besides a sparse-to-dense stream conversion, although it will be a challenge to figure out the right functions.

Superstrider generalizes beyond DRAM.

B. Crossing the valley of death

We designed Superstrider to cross the "valley of death." In our view, an innovation needs immediate product potential and a long term vision to cross the valley:

For an immediate product, Superstrider could be created in an Intel/Altera Stratix 10 MX or Xilinx Virtex Ultrascale+ FPGA, both of which will come with on-package High Bandwidth Memory when they available (for engineering samples) in 6-12 months. They would implement Superstrider in the programmable logic of the FPGA and test the result. Simulations suggest about 50× improvement in speed for an HBM implementation, but the simulations do not account for the speed and energy efficiency penalty inherent in FPGA implementations. Of course, an FPGA test would reduce risk for an ASIC implementation.

Superstrider is part of a project to roadmap the continuation of Moore's law, which recently went down a dead end by limiting itself to the microprocessor and better transistors. The project's goal is to devise and promote architectures like Superstrider that can advance emerging applications areas like graph problems, and then roadmap these approaches in the IRDS to facilitate their funding. The plan is to roadmap a scale-up path like "Moore's law" based on progressively tightening logic-memory interfaces through 3D packaging, as shown in Fig. 2. This paper is one element of the strategy.

REFERENCES

[1] Srikanth, Sriseshan et. al. The same authors with author names in a different order, have submitted a paper to ICRC with details of the archtiecture. If not accepted, this other paper could be published as a Sandia tech report. "[the superstrider paper]." *IEEE ICRC* xx.y (2017): pp. If a reviewer wants to see this paper prior to its publication, it will be avialable temporarily at http://www.debenedictis.org/erik/ComputerPapers/GraphChallenge/ICRC_Superstrider_v3.4.pdf. Also, the simulator is online, open source, at http://www.debenedictis.org/SuperStrider.cpp.

[2] G.E. Moore, "Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, pp. 114 ff," *IEEE J. Solid-State Circuits Newsletter*, vol. 11, no. 5, 2006, pp. 33–35.

[3] International Roadmap for Devices and Systems, http://irds.ieee.org

[4] High Bandwidth Memory (HBM) DRAM (JESD235), JEDEC, October 2013 http://www.jedec.org/standards-documents/results/jesd235

[5] Official website of the Hybrid Memory Cube Consortium http://www.hybridmemorycube.org/

[6] M.M. Sabry Aly et al., "Energy-Efficient Abundant-Data Computing: The N3XT 1,000X," Computer, vol. 48, no. 12, 2015, pp. 24–33.

[7] Batcher, Kenneth E. "Sorting networks and their applications." *Proceedings of the April 30-May 2, 1968, spring joint computer conference*. ACM, 1968.

[8] See https://en.wikipedia.org/wiki/Sorting_algorithm.

[9] Kepner, Jeremy, and John Gilbert, eds. *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011.

[10] Becchetti, Luca, et al. "Efficient semi-streaming algorithms for local triangle counting in massive graphs." Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2008.

[11] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, 2011.

Chapter 4

RRNS for Error Correction

This paper is to appear in the ACM Transactions on Architecture and Code Optimization (TACO).

Extending Moore's Law via Computationally Error Tolerant Computing

BOBIN DENG, Georgia Institute of Technology*
SRISESHAN SRIKANTH, Georgia Institute of Technology*
ERIC R. HEIN, Georgia Institute of Technology
THOMAS M. CONTE, Georgia Institute of Technology
ERIK DEBENEDICTIS, Sandia National Laboratories**
JEANINE COOK, Sandia National Laboratories
MICHAEL P. FRANK, Sandia National Laboratories

Dennard scaling has ended. Lowering the voltage supply (V_{dd}) to sub volt levels causes intermittent losses in signal integrity, rendering further scaling (down) no longer acceptable as a means to lower the power required by a processor core. However, it is possible to correct the occasional errors caused due to lower V_{dd} in an efficient manner, and effectively lower power. By deploying the right amount and kind of redundancy, we can strike a balance between overhead incurred in achieving reliability and energy savings realized by permitting lower V_{dd}. One promising approach is the Redundant Residue Number System (RRNS) representation. Unlike other error correcting codes, RRNS has the important property of being closed under addition, subtraction and multiplication, thus enabling computational error correction at a fraction of an overhead compared to conventional approaches. We use the RRNS scheme to design a Computationally-Redundant, Energy-Efficient core, including the microarchitecture, ISA and RRNS centered algorithms. From the simulation results, this RRNS system can reduce the energy-delay-product (EDP) by about 3× for multiplication intensive workloads and by about 2× in general, when compared to a non-error-correcting binary core.

Extension of Conference Paper: This paper is an extension of "Computationally-Redundant Energy-Efficient Processing for Y'all (CREEPY)" [11]. This submission adds the following:

(1) Correction factor analysis for RRNS signed arithmetic, including an improved correction factor computation for signed multiplication via an LUT based mechanism. (Section 4.8.5)
(2) Design and evaluation of an efficient RRNS multiplier unit by using the index-sum technique, along with associated re-derivation of suitable RRNS bases. (Sections 4.4 and 4.5)
(3) A novel adaptive check insertion strategy that leverages hardware/software runtime or compiler. (Section 4.6.3)
(4) Impact of multi-domain voltage supply to further lower energy consumption. (Sections 4.7, 6.1 and 6.3)
(5) Improved evaluation accuracy by simulating an LLC-main memory hierarchy instead of a perfect cache. (Section 5)
(6) Energy limit analysis for binary, RNS and RRNS cores. (Section 6)

These add significantly more than 30% new material and provide greater insight into RRNS core design. *W.r.t.* written content, every section has been revamped to better present the new findings above.

* These authors contributed equally to this work.
** This work is supported by Laboratory-Directed Research and Development (LDRD) Project 180819 from Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525. Approved for public release SAND2018-0257 J.

DOI: 10.1145/nnnnnnn.nnnnnnn

CCS Concepts: •**Computer systems organization** →**Reliability; Processors and memory architectures; Redundancy;**

Additional Key Words and Phrases: Error-tolerant computing, low energy, reliability

1 INTRODUCTION

Dennard scaling [12] has been one of the main phenomena driving efficiency improvements of computers through several decades. The main idea of this law is that transistors consume the same amount of power per unit area as they scale down in size. However, leakage current and threshold voltage limits caused Dennard scaling to end [46] about a decade ago. This essentially negates any performance benefits that Moore's law may provide in the future; power considerations dictate that a higher transistor density results in either a lower clock rate or a reduction in active chip area.

Theis and Solomon [82] suggest that new device concepts within the purview of two-dimensional lithography technology, such as tunneling FETs, enable reduction of the $\frac{1}{2}CV^2$ energy to small multiples of kT, without resulting in low switching speed [81]. Similarly, research on ferroelectric transistors, *aka* negative capacitance FETs (NCFETs) demonstrates a sub-60mV/dec slope as well as a higher drive current [34–36, 65], both of which are necessary in rendering V_{dd} reduction beneficial to energy reduction without sacrificing performance.

These next generation devices are fast switching even at few tens of millivolts, but as a result, are vulnerable to thermal noise perturbations. This translates into *intermittent, stochastic bit errors in logic*. With signal energies approaching the kT noise floor, future architectures will need to treat reliability as a first-class citizen, by employing efficient computational error correction.

In this paper, we propose a scalable architectural technique to effectively extend the benefits of Moore's law. We enable reducing the supply voltage beyond conservative thresholds by efficiently correcting intermittent computational errors that may arise as a result of thermal noise. Energy benefits are observed as long as the overhead incurred in error correction is less than that saved by lowering V_{dd}. The *creepy* approach of introducing error correcting hardware to lower energy is demonstrated as beneficial in this paper.

1.1 Contributions

(1) Development of microarchitecture, ISA and RRNS centered algorithms towards a Computationally-Redundant, Energy-Efficient core design.
(2) Design and analysis of an efficient RRNS multiplier unit using the index-sum technique.
(3) Novel RRNS-check-insertion heuristics to optimize performance/energy/reliability trade-offs.
(4) Derivation of an estimated lower limit on signal energies via stochastic fault injecting simulation.

We first introduce some mathematical background and notation in Section 2 and then provide a high level overview of a CREEPY core in Section 3 before describing the RRNS algorithms and several other aspects towards designing a CREEPY core in Section 4. We then describe our evaluation methodology, results before discussing related work and concluding in Sections 5, 6, 7 and 8 respectively.

2 BACKGROUND

2.1 Triple Modular Redundancy (TMR)

Error-correcting codes (ECC) are widely used in modern processors to improve reliability. However, these are limited to memory/communication systems and are unable to achieve computational

Table 1. A (4, 2)-RRNS example with the simplified base set (3, 5, 2, 7, 11, 13). Range is 210, with 11 and 13 being the redundant bases.

Decimal	mod 3	mod 5	mod 2	mod 7	mod 11	mod 13
13	1	3	1	6	2	0
14	2	4	0	0	3	1
13+14=27	(1+2)mod 3=0	2	1	6	5	1
	All columns function independently of one another.					
	An error in any one of these columns (residues) can be corrected by the remaining columns.					

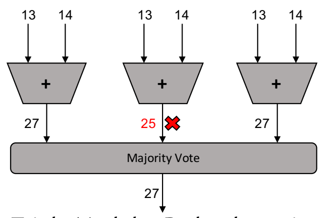


Fig. 1. Triple Modular Redundancy in action.

fault tolerance. The conventional approach to computational fault tolerance is TMR [86]. As shown in Figure 1, the idea is to replicate the computation twice (for a sum total of three computations per computation) and then take a majority vote. With a model that assumes that at most one of these three computations can be in error at any given point in time, it follows that at least two of the computations are error-free; this can thus be used to detect and correct a single error, assuming an error-free voter.

While simple to understand and implement, this introduces more than 200% overhead in area and power leaving plenty of room for improvement. Any energy savings from lowering V_{dd} would be eclipsed due to this overhead in correcting resultant errors.

2.2 Residue Number System (RNS)

The Residue Number System has been used as an alternative to the binary number system chiefly to speed up computation [1, 51]. This increased efficiency comes from the fact that a large integer can be represented using a set of smaller integers, with arithmetic operations permissible on the set in parallel. We present some of the properties of RNS below.

Let $B = \{m_i \in \mathbb{N}\ for\ i = 1, 2, 3, \ldots, n\}$ be a set of n co-prime natural numbers, which we shall refer to as bases or moduli. $M = \prod_{i=1}^{n} m_i$ defines the range of natural numbers that can be injectively represented by RNS that is defined by the set of bases B. Specifically, for x such that $x \in \mathbb{N}\ and\ x < M$, then, $x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, \ldots, |x|_{m_n})$, where $|x|_m = x\ mod\ m$. Each term in this n-tuple is referred to as a residue.

We also note that addition, subtraction and multiplication are closed under RNS. This is because of the following observation: given $x, y \in \mathbb{N}\ and\ x, y < M$, we have $|x\ op\ y|_m = ||x|_m\ op\ |y|_m|_m$, where op is any add/subtract/multiply operation.

2.3 Redundant RNS (RRNS)

To augment RNS with fault tolerance, r redundant bases are introduced. The set of moduli now contains n non-redundant and r redundant moduli: $B = \{m_i \in \mathbb{N}\ for\ i = 1, 2, 3, \ldots, n, n+1, \ldots, n+r\}$. The reason these extra bases are redundant is because any natural number smaller than M $(= \prod_{i=1}^{n} m_i)$ can still be represented uniquely by its n non-redundant residues. Intuitively, the r redundant residues form a sort of *error code* because all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N}$, $x < M$, then,

$x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, \ldots, |x|_{m_n}, |x|_{m_{n+1}}, \ldots, |x|_{m_{n+r}})$ contains n non-redundant residues as well as r redundant residues. For convenience, we further define $M_R = \prod_{i=n+1}^{n+r} m_i$.

Upon applying arithmetic transformations to an RRNS number, any error that occurs in one of the residues is contained within that residue and does not propagate to other residues. When required, such an error can be corrected with the help of the remaining residues. Specifically, an RRNS system with $(n, r) = (4, 2)$, a single errant residue can be corrected, or, two errant residues can be detected. Table 1 provides a simple example, Section 4.8 outlines necessary algorithms to do so. Research by Watson and Hastings [25, 89, 90] lays the foundation for the underlying theoretical framework that is used and extended in our work. Their work also details algorithms to handle RRNS scaling and fractional multiplication. They used (199, 233, 194, 239, 251, 509) as the (4, 2)-RRNS system, providing a range $M = 199 \times 233 \times 194 \times 239 \in (2^{31}, 2^{32})$. In Section 4.5, we discuss the methodology and implications of choosing a different set of RRNS bases for the purposes of trading range with overhead.

Not only does a residue number system achieve a higher efficiency due to enhanced bit-level parallelism (also, no carries required for addition), but also that introducing 50% of overhead is sufficient to provide resiliency. As the granularity of an error is that of an entire residue, RRNS is capable of potentially correcting multi-bit errors as well, for *free*.

We design a computer based on these properties.

3 CREEPY OVERVIEW

Given new device concepts that enable device operation at signal energies close to the kT noise floor [34–36, 65, 81, 82], CREEPY aims to achieve lower energy consumption by lowering V_{dd} in such a manner that the intermittent errors that thereby arise are corrected efficiently.

We take note of the compute preserving properties of RRNS (*cf.* Section 2.3) and propose building a Turing-complete computer around this idea.

A CREEPY core consists of 6 *subcores*, an Instruction Register (IR) and a Residue Interaction Unit (RIU), as depicted in Figure 2.

Each subcore consists of an adder, a multiplier, a portion of the distributed register file and a portion of the distributed data cache. The bit-width of these components is same as that of their corresponding residue (8-bit or 9-bit in this example). Each subcore is fault-isolated from the other because it is designed to operate on a single residue of data(analogous to a bit-slice processor, with bolsters). Post a successful instruction fetch (the instruction cache stores instructions in binary, and is ECC-protected), the ECC-checked instruction is dispatched onto the 6 subcores, which then proceed to operate on their corresponding slice of data. For example, adding two registers is done on a per residue basis; the register file is itself distributed across the 6 subcores. Similarly, the data cache is also distributed across the 6 subcores and stores RRNS protected data. The RIU is then responsible to perform any operations that involve more than a single residue.

Section 4 evaluates several aspects of designing such a core, and provides solutions. For example, conventional multipliers incur high cost in both energy and area, therefore, we leverage RRNS properties to provide an efficient solution in Section 4.4. The RRNS base selection is also very important in CREEPY core design because it directly affects the computational range and energy efficiency, which we discuss in Section 4.5. The RIU logic includes 3 parts: RRNS consistency check logic, RRNS comparison logic and RRNS to binary conversion logic.For the RRNS consistency check logic and RRNS comparison logic designs, we have detailed discussions in Section 4.8.1 and Section 4.8.4 respectively. The RRNS to binary conversion logic is relatively less important as it is used only to support operations that are not native to RRNS, such as bit-shifting and division, which are relatively few in number. Furthermore, the circuitry to convert from RRNS to binary is identical to

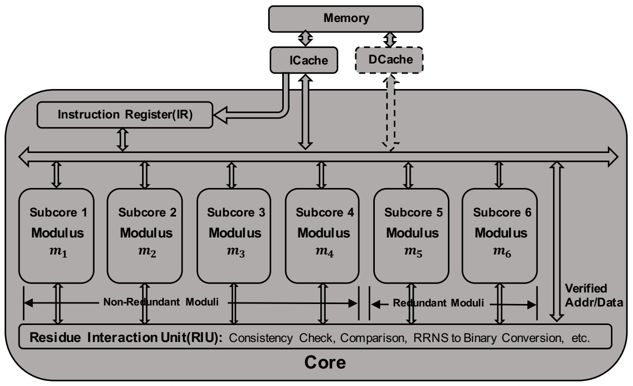


Fig. 2. The CREEPY Core with the reference RRNS system. The register file and data cache are distributed across the subcores.

that found in the literature[7, 25, 76] to convert from RNS to binary, therefore we omit it for space constraints.

Because conversions to and from binary are expensive and rather unnecessary for RRNS data, a CREEPY core operates entirely on RRNS data and literals. An upshot of this is that control-path errors manifest themselves as data errors, meaning that they can be handled simply by handling the data error. For example, if there is an error in bypass logic in a subcore, or, if a faulty decoder in one of the subcores causes it to perform a multiplication instead of an addition, the resultant residue for that subcore would have an erroneous value, but can be recovered from the remaining 5 residues that were a result of the correct addition operation.

Although operating entirely on RRNS data and literals avoids the significant overheads of converting to/from binary, representing a memory address (for the purposes of PC, LD and ST) in an RRNS format naively may cause significant degradation in locality and changes memory access patterns, which is fundamental to memory systems performance. This issue has already been handled by Srikanth et. al. [70], where they propose bit-manipulation techniques as well as a compiler based approach, with little to no overhead. Of the techniques proposed, their rns_sub scheme, which essentially subtracts the least significant residue from the others, renders the most energy efficient architecture along with the added advantage of requiring no support from the software stack.

Interfacing a CREEPY core with heterogeneous accelerators (such as those on an SoC) that may or may not be RRNS based, is a more involved issue, and we leave that to future work.

CREEPY employs standard ECC-protected main memory because of ECC's compactness and efficiency when it comes to protecting stored data. However, standard ECC isn't amenable to computational fault tolerance and therefore, the representation of data is in RNS form (as opposed to binary). The memory controller checks ECC on a processor load and generates the two redundant residues before loading the resultant RRNS data into the last level cache. Similarly, it generates ECC upon a processor store (and the redundant residues are not stored into the main memory). The exact choice of ECC is not relevant to this article; any of the existing schemes [43] may be used.

4 CREEPY CORE

In this section, we present several considerations for the design of the CREEPY core.

4.1 Instruction Set Architecture (ISA)

The description of CREEPY ISA is laid out in a manner similar to that of the MIPS ISA, for explanatory purposes. To simplify instruction fetch and decode, all instructions are of fixed length; 32 bits. The

ISA expects 32 registers (R0-R31), with R0 hard-wired to zero, R30 being the link register and R31 storing the default next PC (= *PC* + 4). In our micro-architecture, each register is 49 bits long (*i.e.,* it contains the RRNS redundant residues as well) and is sliced on a per-modulus (sub-core) basis. The data cache is also implemented in a similar manner, as it stores data in an RRNS format.

(1) **R-Format (ADD/SUB/MUL)**

These instructions assume that the destination operand as well as both source operands are registers.

Opcode	Src Reg1	Src Reg2	Dest Reg	Reserved
6b	5b	5b	5b	11b

(2) **I-Format (ADDI/SUBI/MULI)**

For instructions that require compiler generated immediate literals, two new instructions (that always occur in succession without exception) are defined. Telescopic op-codes are employed to facilitate implementation of such *set* instructions. The fundamental need for the *set* instruction arises from the fact that literals are 49 bit RRNS values and would not otherwise simply fit within a 32 bit field (next to an immediate instruction, for example).

Set123 sets the the first 3 residues of the immediate value into the first 3 sub-core slices of the destination register and *Set456* sets the remaining 3 residues of the immediate value into the other three sub-core slices of the destination register.

Opcode	Dest	Reserved	Residue3	Residue2	Residue1
11[2b]	5b	0[1b]	8b	8b	8b

Opcode	Dest	Residue6	Residue5	Residue4
11[2b]	5b	9b	8b	8b

For an example, consider the immediate instruction *Addi R1, R2, 0x020202020202*. A CREEPY program would implement this instruction as follows:

(a) Set123 R3, 020202
(b) Set456 R3, 020202
(c) Add R1, R3, R2

(3) **Branch**

Opcode	Reg1	Reg2	Reg3	Link	Reserved
6b	5b	5b	5b	1b	10b

Recall that R0 = 0, R31 = PC + 4 and that R30 is the link register. A CREEPY branch follows one of the following semantics:

(a) Reg1 = R0 and Reg3 = R0 and Link = 0: An unconditional branch that always jumps to the address in Reg2.
(b) Link = 0: A conditional branch that jumps to the address in Reg2 (base) + Reg3 (offset) if Reg1 is 0. This is otherwise known as a *beqz* instruction.
(c) Link = 1: A branch and link instruction to enable sub-routine calls and returns. The default next PC is stored into the link register and the program jumps to the address in Reg2.

(4) **Load/Store**

Opcode	Reg1	Reg2	Reg3	Reserved
6b	5b	5b	5b	11b

Reg3 is the destination for a load and is the source register for a store. The source/destination address for a load/store is given by Reg1 (base) + Reg2 (offset). Note that the memory address is hereby stored in an RRNS format. Recall from Section 3 that efficiently handling RRNS addresses without conversion to binary is critical to application performance. Tradeoffs and methodologies in this space have been handled by Srikanth et. al. [70].

(5) **RRNS Check**

Opcode	Reg1	Reserved
6b	5b	21b

Reg1 is the register that needs to be checked. Once an error is detected, the system would try to correct it, for example, by performing the RRNS Single Error Detection and Correction algorithm (Section 4.8). Candidate usage scenarios are discussed in Section 4.6 and evaluated in Section 6. Helper instructions such as *mov*, *ret* etc. also exist, but are omitted from this description for brevity.

4.2 Error Model

First, we distinguish fault, error and failure as follows:

Fault. A single bit flips, but is not stuck-at, *i.e.,* only intermittent / transient faults are considered. Causes may range from unreliable devices to low supply voltage to particle strikes to random noise and any combination therein.

Error. One or more faults in a single residue that show up during a consistency check.

Failure. Error uncorrectable and no recovery mechanism, or error undetectable.

Faults may lead to errors which may lead to failures. *We can guarantee the system is reliable if at most one error per core occurs between two RIU checks.* Multiple bit flips are rare but this phenomenon occurs if a circuit in the carry chain fails [40]. In our design, carry chains are limited to a residue as there are no carries between residues. Therefore, any resulting multi-bit errors would be localized to a single residue, which we can correct. If this RRNS system needs to detect and correct multiple error residues, an extra checkpoint and rollback mechanism is necessary. However, based on the discussion above, the case of multiple residues in error is extremely rare. So we ignore the checkpoint mechanism design in current system and leave it to future work.

Redundancy in time, *i.e.*, check at cycle x, check again at cycle y, check again at cycle z, and vote, does not apply to this model as it is possible that the three checks suffer 3 independent 1 bit faults, rendering voting useless. The *transient* clause in the model rules out stuck-at faults. An implication of this is that we cannot achieve reliability by merely trading performance alone. Additional resources in terms of spatial redundancy are necessary, which is exactly what has been designed.

Different components of the core are protected via specialized means that target each component. The guiding principle is to design a system that uses the more efficient of RRNS/ECC based redundancy based on the range and nature of data being protected. Where both techniques are deemed insufficient to prevent the fault from metastasizing into an error, and eventually into a failure, the more conventional (and expensive) method: Triple Modular Redundancy (TMR), is employed. An alternative is to prevent the fault from occurring in the first place by using high V_{dd} (and/or circuit hardening). Choosing optimally between the latter expensive techniques is beyond the scope of this document but we assume that the RIU uses a high V_{dd} / hardened circuitry. We assume that error in control signals manifest themselves as errors in data (for example, a control error causing one of the subcores to operate on the wrong opcode will be caught as a data error); however, one can potentially further improve the control signals' integrity by using either TMR or intelligent state assignment, and that the RIU uses a high V_{dd} / hardened circuitry.

4.3 Signed Number Representation

There are three competing ways of representing signed numbers, given an RRNS framework as presented in Section 2.3. Each presents its set of trade-offs, which we now detail.

M is the product of all the non-redundant moduli (M = m1*m2*m3*m4) and MR is the product of all the redundant moduli (MR = m5*m6).

(1) **Complement M*MR Signed Representation**

The M*MR complement signed representation is depicted by Figure 3. To provide a few examples, 0 is represented by 0, 1 is represented by 1, $\frac{M}{2} - 1$ is represented by $\frac{M}{2} - 1$, -1 is represented by $M * MR - 1$ and $-\frac{M}{2}$ is represented by $M * MR - \frac{M}{2}$. This is similar to signed binary representation. However, representing numbers in this manner breaks known error correction algorithms[89].

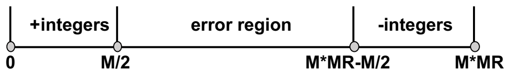


Fig. 3. Complement M*MR signed representation

(2) **Complement M Signed Representation**

The M complement signed representation is depicted in Figure 4. This is similar to the M*MR complement representation, except that the wrap-around occurs at M as opposed to M*MR. This representation does not break error correction algorithms, provided that some correction factors (scaling and offset) are applied to the result of each arithmetic operation. However, further analysis indicates that these correction factors require knowledge of the signs of the operands, which are not trivial to determine like in binary. The RRNS sign determination is a time-consuming algorithm. Moreover. arithmetic operation overflow detection is unknown for this representation.

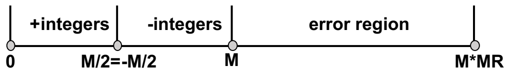


Fig. 4. Complement M signed representation

(3) **Excess-$\frac{M}{2}$ Signed Representation**

The Excess-$\frac{M}{2}$ signed representation is depicted in Figure 5. The excess notation, sometimes known as offset notation, merely shifts each number by $\frac{M}{2}$. To further elaborate, 0 is represented by $\frac{M}{2}$, 1 is represented by $\frac{M}{2} + 1$ and -1 is represented by $\frac{M}{2} - 1$. Similar to the M Complement representation, the results of arithmetic operations must be offset by a correction factor before they can be corrected. However, these correction factors turn out to be independent of the sign of the operands. We also find that this representation enables simple algorithms for comparison (and thereby sign detection) and arithmetic operation overflow detection. In fact, these algorithms make use of a technique used in the error correction algorithm itself. These algorithms are discussed in detail in Section 4.8.

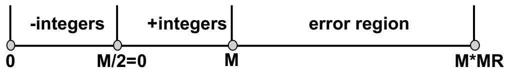


Fig. 5. Excess -M/2 signed representation

We choose Excess-$\frac{M}{2}$ to be the de facto signed representation scheme for CREEPY.

4.4 Optimized Multiplier Unit Design

Many workloads in the domains of multimedia, image processing and digital signal processing are highly multiplication intensive [87] . Index-sum multiplication has been proposed in the past [57, 58] to achieve multiplication via simple addition and table lookup operations, thereby rendering it more efficient than traditional binary multiplication provided the size of the LUT is not too large. The principle is analogous to using a *logarithm* operation, *i.e.,* a multiplication can be achieved via a table lookup, addition and a reverse table lookup, as summarized as follows for the product of two numbers X and Y:

Table 2. Mapping table of $GF(59)$ with a primitive root of 11 ($g = 11$)

X	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
α	0	7	2	14	42	9	10	21	4	49	1	16	25	17	44	28	48	11	34	56
X	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**
α	12	8	47	23	26	32	6	24	22	51	53	35	3	55	52	18	37	41	27	5
X	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**		
α	40	19	57	15	46	54	45	30	20	33	50	39	38	13	43	31	36	29		

Ex: $X = 3$, $Y = 6 \implies \alpha_X = 2$, $\alpha_Y = 9$, whose sum is 11, which reverse maps to 18 and is indeed the desired product.

(1) Use a pre-defined mapping table to generate $index(X)$ and $index(Y)$.
(2) Compute the sum $Z = index(X) + index(Y)$.
(3) Use a pre-defined reverse mapping table to return the product XY as $reverse\_index(Z)$.

While realizing an LUT that is addressable by a 32-bit input is rather expensive, leveraging RNS properties allows us to slice this table into a few tables with address sizes closer to 8 bits, as outlined by Preethy *et al.* [57, 58] . We extend this idea into RRNS by adjusting the RRNS bases (*cf.* 4.5) to be amenable to index-sum LUTs, the requirements for which, are summarized below.

Index-sum multiplication is based on the theory of Galois fields, which can be classified into 3 types: $GF(p)$, $GF(p^m)$ and $GF(2^m)$, where, p is an odd prime number and $m \in \mathbb{Z}^+$. The range of integers that can be represented bijectively in Galois fields, and the encoding methodology depends on the GF type[58]: (We skip the methodology of deriving $GF(p^m)$ as we don't utilize this for CREEPY.)

$GF(p)$: Any integer $x \in [1, p-1]$ can be uniquely coded as a single integral index code α by the relationship $X = |g^\alpha|_p$, where $\alpha \in [0, p-2]$, and g is a primitive root such that $|g^{p-1}|_p = 1$. See Table 2 for an example.

$GF(2^m)$: Any integer $x \in [1, 2^m - 1]$ can be coded as a triple integral index code $< \alpha, \beta, \gamma >$ by the relationship $X = 2^\alpha |5^\beta (-1)^\gamma|_{2^m}$, where $\alpha \in [0, m-1]$, $\beta \in [0, 2^{m-2} - 1]$ and $\gamma \in [0, 1]$. See Table 3 for an example.

Therefore, the relative preference of GF types are $GF(p) > GF(p^m) > GF(2^m)$ as they require 1, 2 and 3 index codes respectively. Furthermore, a smaller value of p and m leads to a smaller LUT. These considerations impact the choice of RRNS bases, as discussed in Section 4.5 / Table 4.

By using the index-sum technique in conjunction with RRNS, we greatly simplify the complexity of multiplication. Index-sum multiplication can be efficiently performed via a simple addition and two modest table lookup operations. We achieve a reduction in ALU gate count using this approach by about 87% when compared to using a traditional multiplier in RRNS, which itself reduces the gate count by 52% when compared to a traditional non-error-correcting binary ALU, thereby realizing area, energy and reliability improvements, as we demonstrate in Section 6.

4.5 Selecting RRNS Bases

Watson [89] used the base set (199, 233, 194, 239, 251, 509) in his paper. However, the range rendered by this set is larger than 2^{31} but smaller than that of a 32-bit unsigned integer: 2^{32}. Furthermore, these bases are not amenable to designing index-sum based multipliers, as discussed in Section 4.4. These limiting necessary and sufficient conditions can be summarized as follows:

(1) Each pair of bases m_i, m_j must be relatively prime. (For RRNS representation [89].)
(2) $max_{n+1 \le i \le n+r} \frac{M_R}{m_i} \ge max_{1 \le i \le n} m_i$
(3) $M_R \ge max_{1 \le i \ne j \le n} m_i m_j$
(4) $M_R \ne 2m_i m_j - n_1 m_i - n_2 m_j$; $1 \le i \ne j \le n; 1 \le n_1 \le m_j - 1; 1 \le n_2 \le m_i - 1$
(5) $M_R \ge 2 \sum_{i=1}^{n}(m_i - 1) + \sum_{i=n+1}^{n+r}(m_i - 1)$. (For RRNS single error correction [89].)

Table 3. Mapping table of $GF(2^6)$

X	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
α, β, γ	0,0,0	1,0,0	0,3,1	2,0,0	0,1,0	1,3,1	0,10,1	3,0,0	0,6,0	1,1,0	0,5,1	2,3,1
X	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
α, β, γ	0,15,0	1,10,1	0,4,1	4,0,0	0,12,0	1,6,0	0,7,1	2,1,0	0,13,0	1,5,1	0,14,1	3,3,1
X	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
α, β, γ	0,2,0	1,15,0	0,9,1	2,10,1	0,11,0	1,4,1	0,8,1	5,0,0	0,8,0	1,12,0	0,11,1	2,6,0
X	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**
α, β, γ	0,9,0	1,7,1	0,2,1	3,1,0	0,14,0	1,13,0	0,13,1	2,5,1	0,7,0	1,14,1	0,12,1	4,3,1
X	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**
α, β, γ	0,4,0	1,2,0	0,15,1	2,15,0	0,5,0	1,9,1	0,6,1	3,10,1	0,10,0	1,11,0	0,1,1	2,4,1
X	**61**	**62**	**63**									
α, β, γ	0,3,0	1,8,1	0,0,1									

Ex: $X = 3$, $Y = 6$ map to, respectively, $< 0, 3, 1 >$, $< 1, 3, 1 >$, whose sum results in $< 1, 6, 2 >$. Since $\gamma \in [0, 1]$, the modulo sum results in $< 1, 6, 0 >$, which reverse maps to 18 and is the desired product.

Table 4. New base sets that satisfy all conditions listed in Section 4.5. For reference, the range of Watson's base set (8-8-8-8-8-9) is 2,149,852,322, and 2^{32} is 4,294,967,296.

Subcore bits	Total bits	Range	Possible base sets	Bases' format
6-8-7-7-8-8	44	82,600,832	**(61,149,128,71,179,181)**	(p,p,2^7,p,p,p)
7-8-7-8-8-8	46	467,921,792	(97,223,128,169,239,241)	(p,p,2^7,13^2,p,p)
7-8-7-8-8-9	47	729,405,056	(113,239,128,211,251,263)	(p,p,2^7,p,p,p)
8-8-7-8-8-8	47	635,871,872	(151,167,128,197,211,223)	(p,p,2^7,p,p,p)
7-8-8-7-9-9	48	430,002,432	(89,233,256,81,283,293)	(p,p,2^7,3^4,p,p)
8-8-8-8-8-9	49	2,149,852,322	**(199,233,194,239,251,509)**	Watson[89]**
9-6-9-5-10-10	49	251,904,512	(269,59,512,31,521,523)	(p,p,2^9,p,p,p)
9-7-8-8-9-9	50	1,230,080,256	(433,81,256,137,439,443)	(p,3^4,2^7,p,p,p)
9-7-9-7-10-10	52	1,719,885,312	(367,113,512,81,521,523)	(p,p,2^9,3^4,p,p)
9-8-8-9-9-10	53	7,891,035,392	**(421,211,256,347,503,521)**	(p,p,2^8,p,p,p)
9-9-8-9-9-9	53	7,710,332,672	(277,317,256,343,409,421)	(p,p,2^8,7^3,p,p)

** these bases do not satisfy index sum constraint.

(6) $|m_1 m_2 - m_3 m_4| = 1$; also known as the $K, K-1$ property. (For RRNS fractional multiplication [89].)

(7) $m_i \in$ {x | x is either (p) prime, (p^m) a power of prime, or (2^m) a power of 2}. Recall that the relative order of preference is $p > p^m > 2^m$, and that smaller bases result in smaller ROMs. (For index sum multiplication (Section 4.4).)

The proofs of Condition (1)-(6) are available in Waston's thesis[89] and Condition (7) is based on the theory of Galois fields which has been discussed in Section 4.4. We limit our analysis to $(n, r) = (4, 2)$ for simplicity and find bases that satisfy the conditions summarized above, while keeping the overhead to a minimum. Table 4 lists several such possibilities. (61, 149, 128, 71, 179, 181) is the set of bases that offers least overhead, whereas (421, 211, 256, 347, 503, 521) on the other hand, offers a range superior to 2^{32} at additional overhead.

4.6 RRNS Check Insertion Strategies

Given that the CREEPY microarchitecture supports the error model outlined in Section 4.2, it is necessary to carefully insert RRNS_check instructions as they have a direct impact on the performance-energy-reliability metrics of the core. In this section, we outline the following check insertion schemes.

4.6.1 ***Periodic check.***

Insert a single check instruction after every n instructions. When $n = \infty$, this is an unchecked core and when $n = 1$, every instruction is checked. Note that lowering the value of n increases the check insertion frequency, raising performance overhead. While increased check insertion frequency typically provides increased reliability, one must be vary of the fact that the check instruction itself is of non-zero latency (*cf.* Section 4.8), meaning that, the longer the core spends in consistency checking, the longer it leaves its state vulnerable for errors to *creep* in. On the other hand, not checking every instruction also increases the probability of errors manifesting into multiple residues, leading to core failure.

4.6.2 ***Pipelined check.***

Insert a pipelined check that checks n instructions after every n instructions. This approach has the performance advantage of amortizing the latency of RRNS_check via pipelining as well as the reliability advantage of being able to increase state coverage of consistency check.

4.6.3 ***StateTable guided adaptive check.***

We define a bookkeeping entity known as $StateTable$ in Section 5 to maintain temporal information of the vulnerability of processor state. Whenever the probability of a register exceeds a certain threshold, an RRNS_check is inserted for that register. Naturally, this can be extended to insert pipelined checks if more than one register is in need of a check. This $StateTable$ itself is assumed to be an error-free entity that can either be implemented in software or hardware. Like the other two schemes, this insertion scheme can be implemented by the compiler or by the runtime (hardware or software); however, it is likely that utilizing a runtime component for this purpose would yield greater accuracy, which translates to improved efficiency and reliability, although subject to the overhead the $StateTable$ itself introduces.

Irrespective of the check strategy, we acknowledge that the following need to be error-free for correct execution; however, for the purposes of this simulation, we ignore their overheads/implications on control flow by assuming periodic checkpointing for potential rollbacks: (1) Effective address of each memory access (RRNS check), (2) Instruction contents (standard ECC check), and, (3) Main memory contents (standard ECC check). For a low-overhead checkpoint mechanism candidate, one can use an incremental checkpoint scheme to save energy and reduce storage overhead, when compared to using full checkpoints alone. The incremental checkpoints only record the modified entries from the last checkpoint (the last checkpoint could either be a full checkpoint or an incremental checkpoint). Once the rollback operation is necessary, the system can then use the last full checkpoint and the subsequent incremental checkpoints to recovery the machine state. A detailed trade-off analysis of the size, frequency, reliability and energy of such a scheme is beyond the scope of this paper.

4.7 Multi-Domain Voltage Supply

The error distribution for each domain of a CREEPY core, *viz.*, computational logic, SRAM cells and RIU logic, are different. In an SRAM device, any fault occurring in one of its transistors gets latched, thereby resulting in an error. To contrast, glitches in logic transistors get masked if the glitch does not occur close to the clock edge. Also, to avoid having to 'check a check instruction', we assume the RIU logic is error-free protected via TMR, hardened logic, and/or higher signal energies, with the latter sufficient to model the energy effects of the former. Given that the vulnerability of these domains increases from computational logic to SRAM cells to RIU logic, it is inefficient to assume a uniformly high signal energy across these domains. We model this phenomena by independent voltage rails for each of these domains. Shimazaki [68] and Rusu [64] proposed some multi-voltage domain designs. The voltage domains referred to in CREEPY are coarse-grained (module-based), rendering the implementation feasible.

Table 5. Error Correction table of RRNS System with Moduli (3,5,2,7,11,13)

Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ
1 , 10	4 6	4 , 5	1 1	5 , 12	3 1	7 , 7	4 3	9 , 3	2 2
2 , 10	2 3	4 , 6	4 4	6 , 1	3 1	7 , 8	1 2	10 , 3	4 1
2 , 12	4 6	4 , 7	2 1	6 , 4	2 4	8 , 1	2 2		
3 , 3	1 1	4 , 11	4 5	6 , 5	4 3	8 , 4	4 2		
3 , 9	4 5	5 , 8	4 4	7 , 2	4 2	8 , 10	1 2		
3 , 12	2 3	5 , 9	2 1	7 , 6	4 2	9 , 1	4 1		

4.8 RIU Algorithms

The algorithm for single error correction was originally given by Watson [89]. However, RNS renders comparison and arithmetic overflow detection to be a non-trivial exercise. We present algorithms to perform these RIU functions by augmenting the consistency checking algorithm. This way, no extra hardware is warranted beyond that required by the error check.

4.8.1 ***Single Error Detection and Correction Algorithm***. The single error detection and correction algorithm proposed by Watson [89] is based on an error correction table. The working of this algorithm for a system with 4 non-redundant moduli (m_1, m_2, m_3, m_4) and 2 redundant moduli (m_5, m_6), for any given integer X ($< M = m_1m_2m_3m_4$) is as follows: (a) Use a base-extension algorithm [25, 52, 89] to compute $|X'|_{m_5}$ and $|X'|_{m_6}$, where $|X'|_{m_5}$ and $|X'|_{m_6}$ are the outputs of the parallel RIU base-extension algorithm. The inputs to the RIU base-extension algorithm are the outputs of non-redundant subcores: $|X|_{m_1}$, $|X|_{m_2}$, $|X|_{m_3}$ and $|X|_{m_4}$, where $|X|_m = X \ mod \ m$; the computational output of the subcore with m_i modulus. (b) For $i = 5, 6$: compute $\Delta m_i = |X'|_{m_i} - |X|_{m_i}$. (c) A non-zero difference indicates the presence of an error. This pair of differences indexes into an entry of a pre-computed (fixed) error correction table, which contains the index of the residue that is in error and a correction offset that needs to be added to that residue to correct said error.

The RRNS_check instruction performs this RRNS Single Error Detection and Correction algorithm. For the error detection step, the system would perform (a) and (b) to the get values of Δm_5 and Δm_6. For the error correction step (if necessary), it performs (c). Analysis of the algorithm reveals that the error detection step would take 8 cycles while the correction step takes 2 cycles. Therefore, once the system inserts an RRNS_check instruction, the first step is to execute the 8-cycle error detection procedure. If no error is found, then this RRNS_check instruction is complete and it takes 8 cycles in total. But if an error is detected, then we need 2 more cycles for the RRNS correction operation to complete (resulting in 10 cycles in total).

For ease of presentation, we present such an error correction table for a smaller (toy) set of RRNS base moduli in Table 5. The total entries in such a table is at most $2\sum_{i=1}^{4}(m_i - 1)$. For the remainder of this section, these set of bases are used for explanatory purposes.

4.8.2 ***Unsigned Number Overflow Detection***. In the absence of any error or overflow, adding 2 unsigned RRNS numbers results in both Δm_5 and Δm_6 being zero. As has been just explained, presence of an error is handled by the error correction table. In the absence of error, we observe that any overflow manifests itself as a fixed index into the error correction table, with the entry not corresponding to any error. Table 6 provides some examples of this observation. While computation of the deltas is most efficient using a base-extension algorithm, we use Chinese Remainder Theorem(CRT) or the Mixed-Radix Conversion (MRC) method to first convert the RRNS number to binary, before computing deltas. This is solely for explanatory purposes; binary conversion is not actually necessary to detect overflow.

Iterating through all possible combinations of numbers and operations, we observe that the value pair of (Δm_5, Δm_6) is fixed. Moreover, (Δm_5, Δm_6) = (10,11) is not a legitimate address of the

Table 6. Unsigned Number Overflow Examples in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	CRT/MRC	$\lvert X'\rvert m_5,\lvert X'\rvert m_6$	Δm_5, Δm_6
2+209	(2,2,0,2,2,2)	(2,4,1,6,0,1)	(1,1,1,1,2,3)	(1, 1, 1, 1) $\Leftrightarrow$ 1	$\lvert 1\rvert_{11}$=1,$\lvert 1\rvert_{13}$=1	10 11
3+209	(0,3,1,3,3,3)	(2,4,1,6,0,1)	(2,2,0,2,3,4)	(2, 2, 0, 2) $\Leftrightarrow$ 2	$\lvert 2\rvert_{11}$=2,$\lvert 2\rvert_{13}$=2	10 11
...	...	...	...	...	...	10 11
209+209	(2,4,1,6,0,1)	(2,4,1,6,0,1)	(1,3,0,5,0,2)	(1, 3, 0, 5) $\Leftrightarrow$ 208	$\lvert 208\rvert_{11}$=10,$\lvert 208\rvert_{13}$=0	10 11

Table 7. Excess-$\frac{M}{2}$ Overflow Examples for addition of two positive numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\lvert X'\rvert m_5,\lvert X'\rvert m_6$	Δm_5,Δm_6
1+104	(1,1,0,1,7,2)	(2,4,1,6,0,1)	(0,0,1,0,7,3)	(0,0,0,0,1,2)	(0, 0, 0, 0) $\Leftrightarrow$ 0	$\lvert 0\rvert_{11}$=0,$\lvert 0\rvert_{13}$=0	10 11
2+104	(2,2,1,2,8,3)	(2,4,1,6,0,1)	(1,1,0,1,8,4)	(1,1,1,1,2,3)	(1, 1, 1, 1) $\Leftrightarrow$ 1	$\lvert 1\rvert_{11}$=1,$\lvert 1\rvert_{13}$=1	10 11
...	...	...	...	...	...	...	10 11

Table 8. Excess-$\frac{M}{2}$ Overflow Examples for addition of two negative numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\lvert X'\rvert m_5,\lvert X'\rvert m_6$	Δm_5,Δm_6
-1-105	(2,4,0,6,5,0)	(0,0,0,0,0,0)	(2,4,0,6,5,0)	(2,4,1,6,10,12)	(2, 4, 1, 6) $\Leftrightarrow$ 209	$\lvert 209\rvert_{11}$=0,$\lvert 209\rvert_{13}$=1	1 2
-3-104	(0,2,0,4,3,11)	(1,1,1,1,1,1)	(1,3,1,5,4,12)	(1,3,0,5,9,11)	(1, 3, 0, 5) $\Leftrightarrow$ 208	$\lvert 208\rvert_{11}$=10,$\lvert 208\rvert_{13}$=0	1 2
...	...	...	...	...	...	...	1 2

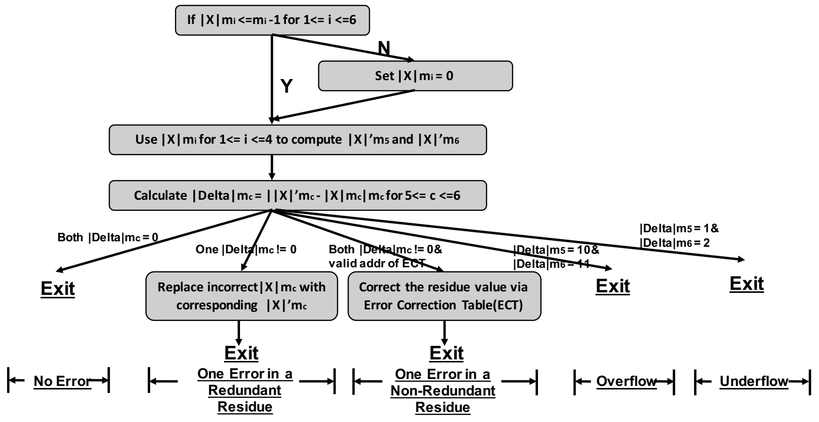

If |X|mi <=mi -1 for 1<= i <=6
N
Y
Set |X|mi = 0
Use |X|mi for 1<= i <=4 to compute |X|'m5 and |X|'m6
Calculate |Delta|mc = ||X|'mc - |X|mc|mc for 5<= c <=6
Both |Delta|mc = 0
One |Delta|mc != 0
Both |Delta|mc != 0& valid addr of ECT
|Delta|m5 = 10& |Delta|m6 = 11
|Delta|m5 = 1& |Delta|m6 = 2
Exit
Replace incorrect |X|mc with corresponding |X|'mc
Correct the residue value via Error Correction Table(ECT)
Exit
Exit
Exit
Exit
No Error
One Error in a Redundant Residue
One Error in a Non-Redundant Residue
Overflow
Underflow

Fig. 6. Single error detection and correction algorithm with overflow/underflow detection

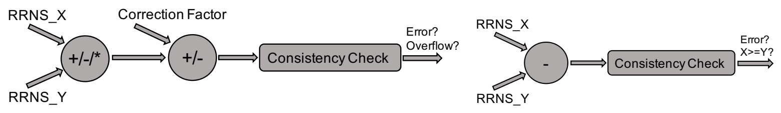
error correction table (Table 5), thus enabling a distinction between an error and an overflow. This approach, however, does not apply to multiplication.

4.8.3 ***Signed Number Overflow Detection.*** Recall from Section 4.3 that CREEPY uses the Excess-$\frac{M}{2}$ signed representation. We discuss the two sources of overflow independently:

(1) *Add two positive numbers.* Table 7 provides a few examples illustrating the algorithm (Correction factors are explained in detail in Section 4.8.5). The $1 + 104$ in the first column is represented in decimal. After Excess-$\frac{M}{2}$ mapping, the computing equation is transformed to $106 + 209$ since $\frac{M}{2} = 105$ for the toy set of moduli. Therefore, the X RRNS value is the the RRNS of 106 and Y RRNS value is the the RRNS of 209. We observe that the pair (Δm_5,Δm_6) remains at a fixed value (10,11).
(2) *Add two negative numbers.* Similarly, examples for adding two negative numbers are shown in Table 8. In this case, we observe that the pair (Δm_5,Δm_6) is fixed to (1,2).

Note that neither (10, 11) nor (1, 2) are legitimate addresses in Table 5, thereby enabling a distinction between an error and an overflow. However, while this method works for both addition and subtraction, it does not hold for detection of multiplication overflow as the delta-pair is not constant and sometimes indexes into a legal error correction table entry.

Figure 6 shows the overview of the whole algorithm.



(a) Signed Overflow Detection (b) Signed Comparison

Fig. 7. Signed Overflow Detection and Comparison

We observe that the described algorithm works in a similar manner even with the base sets in Table 4. E.g. Waston's bases (199,233,194,239,251,509), an overflow results in a delta-pair of (77, 289), whereas an underflow results in (174, 220). Both these pairs do not index into legitimate entries of the error correction table for these set of bases (*cf.* Appendix E, Watson [89]).

4.8.4 ***Comparison***. Comparison is an important operation because of its use in determining control flow. In a manner similar to overflow detection, we explore potential algorithms to perform RRNS comparison without incurring unnecessary hardware overhead.

Jen-shiun et al. [9] and Omondi [52] proposed number comparison methods for residue numbers based on parity bits. However, a prerequisite of these parity comparison methods is that all moduli are supposed to be odd (in addition to being pair-wise relatively prime). In CREEPY, one of the non-redundant moduli is even (to enable fast fractional multiplication [89]), therefore this approach is not suitable.

Instead, we propose leveraging the error check algorithm itself to check for an overflow post a subtraction: To compare X and Y, perform $X - Y$ and derive the delta-pair $(\Delta m_5, \Delta m_6)$. Then, $X \geq Y$ iff the delta-pair is (0, 0) (*i.e.,* no overflow) and $X < Y$ iff the delta-pair is (174, 220) (*i.e., $X - Y$* results in an underflow).

This new residue number comparison method can be used for both unsigned and Excess-$\frac{M}{2}$ signed numbers. It is easy to understand that this idea is suitable for unsigned residue numbers: if $X < Y$, then $X - Y \notin [0, M)$, thereby resulting in an underflow. For an Excess-$\frac{M}{2}$ signed number X, an injective mapped residue number can be defined as follows: $X_{mapped} = \frac{M}{2} + X$. Therefore, $X \geq Y$ iff $X_{mapped} \geq Y_{mapped}$, which reduces to an unsigned comparison. A caveat to note is that correction factors should not be added for a comparison operation. These are summarized in Figures 7a and 7b.

4.8.5 ***Correction Factors***. In this section, we are concerned with the addition, subtraction and multiplication operations on two numbers that do not generate any overflow. Recall from Section 4.3 that CREEPY uses the Excess-$\frac{M}{2}$ notation, which means that there is a bijective mapping from any number x such that $-\frac{M}{2} < x < \frac{M}{2}$ to $x + \frac{M}{2}$. Because of this offset, arithmetic operations results need to be re-adjusted using what we term as *correction factors*. [However, this has nothing to do with the RRNS error correction operation.]

Addition Consider the addition of two numbers x and y. To represent the mapping, define a and b such that $0 \leq a, b < \frac{M}{2}$ so that there is no overflow.

Case 1: $x, y \geq 0$

Consider $x = a$ and $y = b$. The sum $x + y$ can be represented for each subcore $1 \leq i \leq n + r$ as follows:

$$\left| |\frac{M}{2} + a|_{m_i} + |\frac{M}{2} + b|_{m_i} \right|_{m_i} = \left| |M|_{m_i} + |a+b|_{m_i} \right|_{m_i} \tag{1a}$$

$$= |a+b|_{m_i} \; for \; 1 \leq i \leq n \tag{1b}$$

However, the expected addition result is:

$$\left| \frac{M}{2} + a + b \right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |a+b|_{m_i} \right|_{m_i} \tag{2}$$

It follows that:

(1) $1 \leq i \leq n$ and m_i is odd: Examining equations 1b and 2 imply that no correction factor is necessary.
(2) $1 \leq i \leq n$ and m_i is even: Examining equations 1b and 2 implies that a constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.
(3) $n+1 \leq i \leq n+r$: Examining equations 1a and 2 imply that a constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be subtracted from the result.

Case 2: $x, y < 0$
Setting $x = -a$ and $y = -b$, and re-working equations similar to Equations 1a, 1b and 2 result in correction factors that are identical to Case 1.

Case 3: $x > 0$, $y < 0$ (Without loss of generality.)
Setting $x = a$ and $y = -b$, and re-working equations similar to Equations 1a, 1b and 2 result in correction factors that are identical to Case 1.

Subtraction Due to the symmetric and offset based nature of the Excess-$\frac{M}{2}$ representation, we again present the working of just one of the cases; without loss of generality: $x = a$ and $y = b$. Then, $x - y$ becomes:

$$\left| |\frac{M}{2} + a|_{m_i} - |\frac{M}{2} + b|_{m_i} \right|_{m_i} = |a - b|_{m_i} \tag{3}$$

However, the expected subtraction result is:

$$\left| \frac{M}{2} + a - b \right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |a - b|_{m_i} \right|_{m_i} \tag{4}$$

From examining equations 3 and 4, it follows that:
(1) $1 \leq i \leq n$ and m_i is odd: No correction factor is necessary.
(2) $1 \leq i \leq n$ and m_i is even: A constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.
(3) $n+1 \leq i \leq n+r$: A constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.

Multiplication Again, for brevity, we only present the case where two positive integers are multiplied; without loss of generality: $x = a$ and $y = b$; the product xy becomes:

$$\left| |\frac{M}{2} + a|_{m_i} |\frac{M}{2} + b|_{m_i} \right|_{m_i} = \left| |\frac{M^2}{4} + \frac{(a+b)M}{2}|_{m_i} + |ab|_{m_i} \right|_{m_i} \tag{5}$$

However, the expected multiplication result is:

$$\left| \frac{M}{2} + ab \right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |ab|_{m_i} \right|_{m_i} \tag{6}$$

As residues are typically 8-bit wide, consider a 511 entry LUT per subcore that stores the following:

$$LUT(s) = \left| \frac{M^2}{4} + \frac{(s-1)(M)}{2} \right|_{m_i} \tag{7}$$

From examining equations 5, 6 and 7, it follows that:
(1) $1 \leq i \leq n$ and m_i is odd: No correction factor is necessary.
(2) $1 \leq i \leq n$ and m_i is even: The correction factor can be effected by computing $s = a + b$ and then subtracting $LUT(s)$ from the result of the multiplier.
(3) $n+1 \leq i \leq n+r$: The correction factor can be effected by computing $s = a + b$ and then subtracting $LUT(s)$ from the result of the multiplier.

The correction factors for the addition and subtraction operations require a single, constant addition/subtraction operation, whereas for multiplication, 2 additions/subtractions and a modest table lookup are required. Another advantage of the schemes presented here is that sign determination is not necessary and that they can be performed at the subcore level, without the involvement of the RIU.

5 EVALUATION METHODOLOGY

To measure the performance-energy-reliability trade-off of a CREEPY core, we augment a stochastic fault injection mechanism into a cycle-accurate in-order trace-based simulator. We abstract the notion of using next-generation devices operating at low signal energies (E_s) and the resulting interaction with the kT noise floor into P_e, the probability of an error occurring in a transistor state in any given cycle. E_s, provided as an input to the simulation, is a measure of the signal energy at the input of a transistor; P_e is the probability of a fault occurring at the output of a transistor in any given cycle. The relationship of E_s and P_e can be defined by the following relation: $P_e = exp(\frac{-E_s}{kT})$. From Section 4.7, these inputs are vectors as they denote the signal energies and error probabilities for each voltage domain, however, for explanatory purposes, we present them as scalars for the remainder of this section. Also input to the simulator is the check insertion strategy, as discussed in Section 4.6. Because we are evaluating a very different number system, we simulated an unpipelined microarchitecture with no branch prediction and a 2-level memory hierarchy (LLC-DRAM, with latencies of 12 cycles and 100 cycles for LLC hit and miss respectively) to maintain our primary focus in this paper. Adding more features to our design has been left as future work.

We first introduce a series of error events and their probabilities.

P_e Probability of an error occurring in a transistor state in any given cycle. This is provided as an input to the simulation, as just discussed.

P_{add} Probability of at least a single error in an adder (each sub-core has an adder). If there are N_{add} transistors in an adder, the probability of each of these transistors being free of error is $(1 - P_e)^{N_{add}}$. Therefore, P_{add}=1-$(1 - P_e)^{N_{add}}$. Similarly, P_{sub} and P_{mul} are calculated. For multi-cycle operations, this definition holds as long as the state of each transistor is used exactly once for the operation. This is true for the said operators. Note that this is a conservative (pessimistic) estimate in our evaluation because we ignore any error masking that may potentially occur.

P_{R_i} Probability of at least 1 error being present in a slice (sub-core/residue) of register R_i since its last write. To compute this, we devise a $StateTable$, the i^{th} entry of which holds the tuple (P, $cycle$), where, P is the probability of R_i having atleast 1 error being present in the corresponding residue upon its most recent update at cycle $cycle$. This $StateTable$ is updated for each register write.

For example, consider the register R_0. 1) At cycle 0, the default value of R_0 tuple is (P=0, cycle=0). 2) At cycle 10, assume that we have an ADD instruction: ADD R0, R1, R2, and that it is the first instruction writing to R_0. We then update the tuple value to (Error_Probability_ADD, 10). It is necessary to update the P value here because the error probability of this ADD instruction should be taken into account. P value would then be set back to 0 once an RRNS check is inserted for that register and no error is detected, and then set the current system cycle value to the cycle field. This way, the P field in the $StateTable$ always reflects the probability of that register of having at least 1 error being present in one of its residues, given its most recent update at the cycle field.

Assuming an SRAM implementation of 8-bit wide R_i, the number of transistors is $8 \times 6 = 48$. The probability of R_i being error free is subject to two probabilities: (1) probability of an error-free write, ($P_1 = 1 - StateTable[R_i].P$) and, (2) probability of no error creeping into it since its last write ($P_2 = (1 - P'_e)^{48(c - StateTable[R_i].cycle)}$), where, c is the current cycle and P'_e is the probability of an error occurring in the state of an SRAM transistor. Due to the nature of an SRAM device, any fault occurring in one of its transistors gets latched, resulting in a higher probability of an error (when compared with glitches in logic transistors getting masked if the glitch does not occur close to the clock edge). As such, we assume $P'_e = 100P_e$. Putting it all together, we have $P_{R_i} = 1 - P_1 * P_2$.

$P_{LOAD\ X}$ Probability of at least 1 error being present in the loaded data of address X. This is analogous to P_{R_i}, with the extended $StateTable$ storing an entry for each cache line. As we assume a perfect off-chip (ECC protected) main memory, cache miss repairs are initialized with a zero probability in error, and cache replacement victims' entries are evicted from the $StateTable$. Finally, $P_{LOAD\ X}$ encapsulates the probability of an error in the implicit computation of the address X itself (from its base and offset) during the execution of the load, in addition to the probability of an error in the loaded data from the cache line.

P_{SC} Probability of at least 1 error occurring in a sub-core from the last time it was checked. To illustrate, consider the following add instruction: $ADD\ R_3, R_2, R_1$. Then, at the end of instruction, $P_{SC} = 1 - (1 - P_{add})(1 - P_{R_2})(1 - P_{R1})$.

P_C Probability of exactly 1 error occurring in a CREEPY core from the last time it was checked. This translates to exactly 1 sub-core being in error (where the sub-core error itself may be of multi-bit form; RRNS can tolerate multi-bit flips within a single residue). Therefore, $P_C = 6C_1 \times P_{SC}(1 - P_{SC})^5$, where the combinatorial choose operator nC_r enumerates the number of ways in which r items can be chosen from n distinct items.

P_C^0 Probability of no error in a CREEPY core from the last time it was checked. $P_C^0 = 6C_0 \times (1 - P_{SC})^6 = (1 - P_{SC})^6$.

P_C^{fail} Probability of a CREEPY core failing at any given cycle, since the last time it was checked. The current version of the CREEPY micro-architecture is unable to correct more than 1 error occurring in the core, and assumes a recovery mechanism such as checkpointing is in place. As such, we deem ≥ 2 errors in the core as amounting to a failure. Therefore, $P_C^{fail} = \sum_{2 \leq r \leq 6} 6C_r \times P_{SC}^r (1 - P_{SC})^{6-r} = 1 - P_C^0 - P_C$.

Note that the computation of these error probabilities is done after every instruction (irrespective of the check insertion strategy) for the purposes of bookkeeping such as $StateTable$ update and to estimate the probability of a failure $P_{C,i}^{fail}$ at each time step t_i. We use a typically used reliability metric, Mean Time Between Failure (MTBF) [77], which can be defined as follows: $MTBF = \frac{Total\ Cycles}{CPU\ Frequency \times \sum_i P_{C,i}^{fail}}$. The subscript i in $P_{C,i}^{fail}$ represents the i^{th} instruction of the instruction stream. MTBF also corresponds to mean time to checkpoint recovery.

6 SIMULATION RESULTS

6.1 Signal Energy Limits

From an independent set of simulations of a non-error-correcting core operating on binary data, we find that the minimal signal energy required for ensuring its reliable operation is $48kT$. In our previous design of an error correcting RRNS core [11], we assumed a single voltage domain across computational logic, SRAM cells and RIU logic. Together with a traditional multiplier (i.e., without index-sum), a pipelined check insertion strategy (with a frequency of 5 instructions) and a 16MB LLC, the result is that we can tolerate gate signal energies of $42 - 43kT$, as shown in Figure 8.

However, given the dissimilarity in error distribution across computation, SRAM and RIU (Section 4.7), we consider independent voltage domains for these. For simplicity, we conservatively set the RIU gate signal energy to be $48kT$ (i.e., same as that required for a non-error correcting binary core), although it can be potentially lowered as its functionality is a subset of that of a binary core. We find that the relative impact of energy savings in the RIU is rather limited (Section 6.5), and therefore restrict the RIU gate signal energy to $48kT$ in our evaluations.

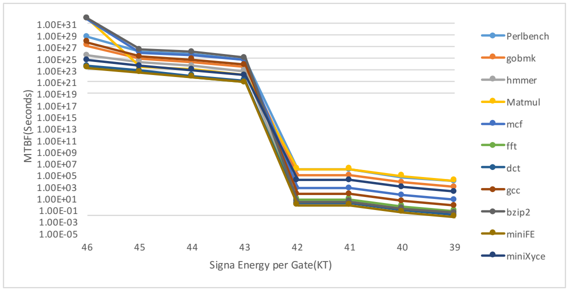


Fig. 8. Reliability when a single voltage domain is used.

Table 9. Sensitivity of MTBF (seconds) to benchmarks and gate signal energies of computational logic, SRAM cells and RIU logic. For example, 30-43-48 denotes the gate signal energy for computational logic to be $30kT$, SRAM cells to be $43kT$, and RIU logic to be $48kT$.

Benchmarks	**36-43-48**	**35-43-48**	**34-43-48**	**33-43-48**	**32-43-48**	**31-43-48**	**30-43-48**	**29-43-48**	**28-43-48**	**27-43-48**
perlbench	1.70E+17	2.29E+16	3.10E+15	4.20E+14	5.69E+13	7.70E+12	1.04E+12	1.41E+11	**1.91E+10***	2.62E+09
gobmk	1.07E+17	1.44E+16	1.95E+15	2.64E+14	3.58E+13	4.85E+12	6.55E+11	8.87E+10	**1.22E+10***	1.97E+09
hmmer	4.08E+16	5.53E+15	7.48E+14	1.01E+14	1.37E+13	1.85E+12	2.51E+11	**3.40E+10***	6.23E+09	9.69E+08
matmul	1.08E+16	1.47E+15	1.99E+14	2.69E+13	3.64E+12	4.93E+11	6.66E+10	**9.02E+09***	1.22E+09	2.76E+08
mcf	1.81E+17	2.44E+16	3.31E+15	4.47E+14	6.06E+13	8.20E+12	1.11E+12	1.50E+11	**2.03E+10***	2.80E+09
fft	7.63E+14	1.03E+14	1.40E+13	1.89E+12	2.56E+11	**3.47E+10***	4.69E+09	6.35E+08	1.86E+08	2.32E+07
dct	1.33E+15	1.80E+14	2.44E+13	3.30E+12	4.47E+11	6.05E+10	**8.18E+09***	1.11E+09	3.11E+08	4.02E+07
gcc	1.41E+17	1.91E+16	2.59E+15	3.50E+14	4.75E+13	6.42E+12	8.68E+11	1.18E+11	**1.60E+10***	2.37E+09
bzip2	1.73E+17	2.34E+16	3.17E+15	4.29E+14	5.81E+13	7.86E+12	1.06E+12	1.44E+11	**1.95E+10***	2.65E+09
miniFE	6.94E+14	9.39E+13	1.27E+13	1.72E+12	2.33E+11	**3.15E+10***	4.26E+09	5.77E+08	1.69E+08	2.11E+07
miniXyce	1.09E+16	1.47E+15	1.99E+14	2.69E+13	3.64E+12	4.93E+11	6.66E+10	**9.02E+09***	2.04E+09	3.06E+08

* We use these signal energies for the remainder of this paper, as they render reasonable reliability.

We abstract these voltage domains as a triplet; for example, $30 - 43 - 48$ denotes the gate signal energy for computational logic to be $30kT$, SRAM cells to be $43kT$, and RIU logic to be $48kT$. For the purposes of this evaluation, we assume a target MTBF of $1E + 10$ seconds (over 300 years) and find that the gate signal energy for computational logic can be lowered all the way to $28 - 31kT$, depending upon the benchmark, as shown in Table 9.

Given these minimum signal energies, we evaluate the performance, efficiency and reliability of various core configurations in Sections 6.2, 6.3 and 6.4 respectively. The core configurations presented are as follows:

Binary A non-error-correcting core operating on binary data. This is the baseline and requires signal energies of at least $48kT$ in order to achieve reasonable reliability.

RNS A non-error-correcting core operating on RNS data. In other words, an RRNS core without redundant subcores and error correction capabilities.

RRNS_pipe5 An error correcting RRNS core with a pipelined check insertion strategy (with a frequency of 5 instructions), as was determined as the most optimal strategy in our previous RRNS core design [11].

Index-sum_pipe5 Similar to RRNS_pipe5, except that the traditional multiplier is replaced with an index-sum multiplier.

RRNS_Adapt_1e-9 An error correcting RRNS core with an adaptive check insertion strategy (with an error probability threshold of $1e-9$. We found that $1e-9$ was the optimal threshold obtained via simulation for target MTBF/signal energy).

Index-sum_Adapt_1e-9 Similar to RRNS_Adapt_1e-9 that uses an index-sum multiplier.

6.2 Performance

Figure 9 presents the performance of various core configurations listed in Section 6.1, normalized to that of a non-error-correcting binary core.

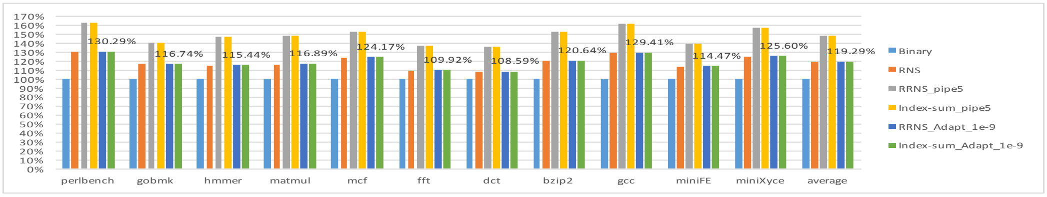


Fig. 9. Performance of various core configurations, normalized to an non-error-correcting binary core

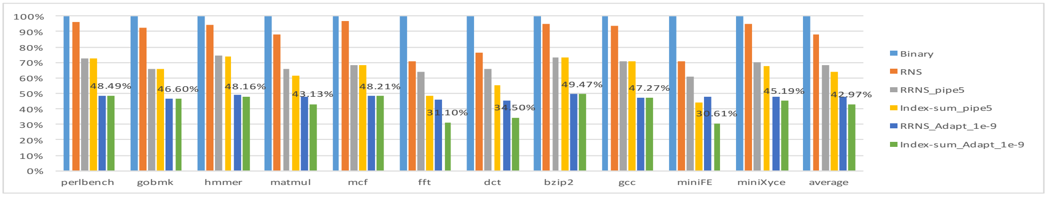


Fig. 10. Energy Comparison for Different Strategies

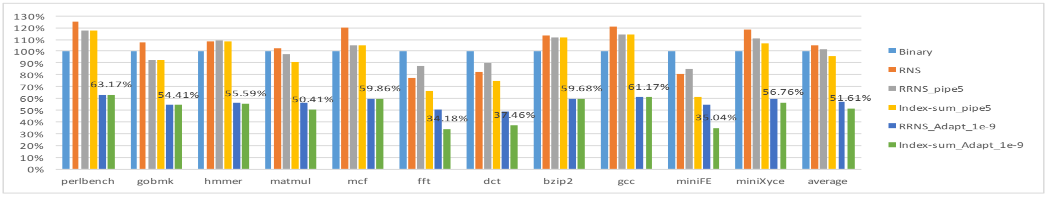


Fig. 11. EDP Comparison for Different Strategies

There is an inherent performance degradation in running binary-optimized code on an (R)RNS-based core because position-based bit manipulation techniques are expensive in (R)RNS, however, this is limited to about 20% on average. Introducing error correction may further degrade performance if naive or static check insertion strategies are used. The overhead due to error correction is amortized when the check insertion strategy is adaptive instead.

6.3 Energy

The primary concern of CREEPY core design is reducing the core energy overhead. Figure 10 shows the normalized energy consumption of the aforementioned configurations.

The non-error-correcting binary core requires high gate signal energies in order to be reliable. Given the low-bit-width and carry-free nature of RNS arithmetic, RNS based cores are inherently more energy efficient than their binary counterparts. When *efficient* error correction is introduced, further energy savings can be achieved as the supply voltage can be turned down while still maintaining reliable functionality. We ensure that the overhead of error correction is minimal by using an adaptive check insertion strategy. Finally, using index-sum multipliers enables further energy savings as they are more efficient than traditional multipliers (savings of over 3× for multiplication intensive benchmarks and over 2.3× on average).

6.4 Energy Delay Product(EDP)

Figure 11 shows the Energy Delay Product (EDP) of these core configurations, normalized to that of a non-error-correcting binary core. With the exception of arithmetic intensive workloads, RNS cores typically have a higher EDP than binary cores. However, via efficient error correction, our RRNS cores show significantly improved EDP. Specifically, by utilizing our best optimization scheme (index-sum multiplier and adaptive check insertion), we see EDP benefits of about 2× on average, or about 3× for multiplication intensive workloads.

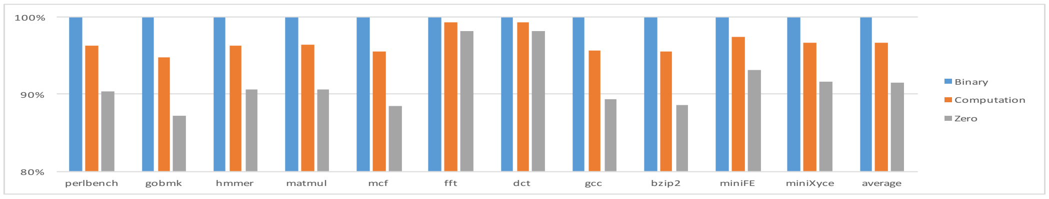


Fig. 12. The Potential of RIU Energy Optimization

6.5 Energy Potential of RIU Optimizations

As described in Sections 4.7 and 6.1, we conservatively choose the gate signal energy for RIU logic to be that necessary for reliable operation of a Turing complete non-error-correcting binary core, i.e., $48kT$. One of the reasons for this is to side-step the issue of 'checking the checker'. However, if we were to deploy self-checking logic or some other optimizations in the RIU, it may no longer be necessary to use a high voltage supply for the RIU domain. In this limits study, we evaluate 3 possibilities of the gate signal energy to RIU logic: *Binary* - $48kT$, *Computation* - same as that of RRNS subcore computational logic, *Zero* - $0kT$. From an Amdahl's law perspective, we find that optimizing RIU logic has limited impact on core energy, as shown in Figure 12, thanks to our judicious RIU usage via adaptive check insertion.

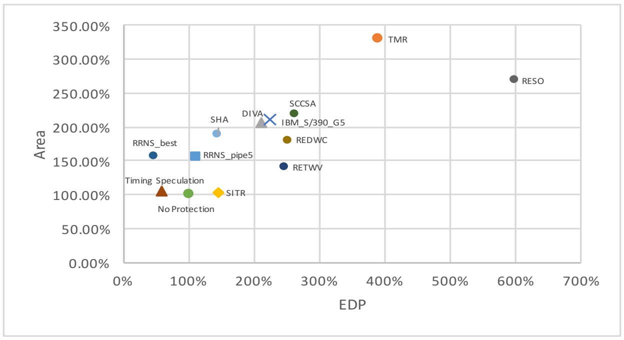
7 RELATED WORK

RNS and RRNS The energy efficient properties of RNS due to its low-bit-width operations and absence of carries across residues has found applications in the digital signal processing (DSP) [10, 14, 60] domain. Furthermore, the representability of high bit-width integers as a tuple-of-resides has been leveraged by the cryptography (RSA) [4, 28, 94] community. Anderson [1] proposed an architecture and ISA for an RNS co-processor designed to run datapath operations in tandem with a general-purpose processor running binary instructions, where the primary role of the general purpose processor is to handle control flow. The RNS co-processor uses an accumulator based ALU and does not support caching or computational error correction (RRNS). Furthermore, it requires a conversion to binary (and vice-versa) for comparison operations, which is expensive. Clearly, our CREEPY architecture is significantly more efficient. A unique feature of their ISA is their ability to encode instructions targeting two ALUs simultaneously. But this can easily be extended to our architecture and enable such Superscalar-like capabilities if need be.

Chiang et al. [9] provide RNS algorithms for comparison and overflow detection, but assume all bases to be odd and do not consider error correction. Similarly, Preethy et al. [57, 58] integrate index-sum multiplication into RNS, but do not consider its impact on the properties of RRNS bases critical to CREEPY.

Ever since Watson and Hastings [25, 89, 90] introduced RRNS as an efficient means for computational error correction, there has been a significant body of research [3, 5, 8, 13, 17, 21, 22, 24, 32, 38, 39, 42, 53, 59, 61, 67, 71–73, 75, 76, 78–80, 91–93, 95] that strives to improve upon it. These are orthogonal to CREEPY, and further such algorithmic research can be used to optimize aspects of the core itself, such as the RIU.

Computational Error Correction Standard error correcting codes (ECC)[43] have already been adopted into modern memory systems. These codes accommodate errors occurring in storage and communication/network traffic, but are not able to protect computational logic. The naive approach to computational error correction is triple modular redundancy (TMR)[86], requiring over a 200% overhead in area and energy for single error correcting capability. Several techniques in the form of arithmetic codes such as AN codes[6, 18, 19, 41, 66, 88], self-checking[30, 33, 44, 48–50, 84] and self-correcting[15, 20, 26, 37, 45, 55, 62, 63, 74, 83] adders and multipliers have since been devised.



RESO[54], REDWC[31], RETWV[27], SCCSA[85], SHA[56], DIVA[2], SITR[47], Timing Speculation[16, 23]

Fig. 13. First order comparison of area overhead and energy-delay product (EDP) of various mechanisms for computational error correction, depicting the superiority of RRNS. Computational error correction techniques use a combination of spatial and temporal redundancy techniques. While temporal redundancy allows for a low area overhead, they suffer from a significant performance penalty. Timing speculation techniques seem more efficient than RRNS, however, their error model assumes all bit errors manifest as circuit timing errors, which is not sufficient to work with ultra low energy logic devices.

Orthogonally, proposals employ redundancy at a higher granularity, such as timing speculation (wherein error correction capability is limited to circuit timing violations)[16, 23], partial pipeline replication[2] or checkpoint-rollback-recovery such as those in IBM Power8 processors[29]. While these are more efficient than naive TMR, they come with limitations on their error model, or, their area overheads are still over 100% and/or incur a significant performance penalty, owing to the fact that they leverage temporal redundancy in an effort to minimize area overhead[69]

Figure 13 summarizes some of these techniques in comparison with RRNS. We refer the interested reader to Srikanth et al.[69] for a more detailed survey on some of these non-residue techniques, but the takeaway is that RRNS is generally considered superior in terms of capability and efficiency for computational error resilience.

Approaches that employ timing speculation[16, 23] may seem superior to RRNS at first glance. However, the error model that can be supported by an RRNS error correcting microarchitecture is orthogonal to theirs, if not broader. For example, razor[16] uses conventional transistors, therefore lowering V_{dd} lowers MOSFET switching speed, resulting in a frequency drop, which could cause setup time violations that they handle via a delayed latch mechanism. They assume that any error manifests itself as a timing error. Similarly, decor[23] uses a delayed commit approach (with rollback support) to handle violations in timing margins. However, with emerging devices (Section 1), V_{dd} can be lowered to few tens of millivolts without frequency loss, meaning that operating at the resultant thermal noise floor leads to *stochastic, intermittent* bit flips, which cannot be captured as circuit timing errors. Unlike such approaches, a CREEPY core can not only tolerate such errors in the data path, but also in the control path between memory accesses.

In terms of being able to tolerate control path errors, approaches such as DIVA[2] that replicate parts of the pipeline are capable. Their design provides recovery by having a simple core recalculate results of an out-of-order core. In this approach, the simple core is assumed to be error-free. This is similar to a ”double-modular-redundancy” approach with a rad-hard node, implying a relatively high overhead. Furthermore, if the rad-hard simple core is instead prone to error, checkpoint and re-execute methods would need to be employed, similar to the IBM POWER7/8 processors[29]. On the other hand, a CREEPY core is able to tolerate errors in its redundant as well as non-redundant computations.

8 CONCLUSION

The advent of next generation device concepts such as tunneling FETs and ferroelectric/negative-capacitance FETs enables reduction of supply voltage to few tens of millivolts without degradation in switching speed. However, as a result of operating close the the kT noise floor, computational logic is subject to intermittent, stochastic errors. The RRNS representation is a promising approach towards using such ultra low power devices, by employing efficient computational error correction.

In this paper, we design a Compuationally-Redundant, Energy-Efficient core, including the microarchitecture, ISA and RRNS centered algorithms. We elucidate several novel optimizations and RRNS-based design considerations to demonstrate significant improvements over a non-error-correcting binary core.

9 ACKNOWLEDGMENT

We sincerely thank the anonymous reviewers of the TACO review process for their constructive feedback on this research.

REFERENCES

[1] Daniel Anderson. 2014. Design and Implementation of an Instruction Set Architecture and an Instruction Execution Unit for the REZ9 Coprocessor System. *M.S. Thesis, U of Nevada LV* (2014).

[2] Todd M Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, MICRO-32. Proceedings. 32nd Annual International Symposium on.* IEEE, 196–207.

[3] Jean-Claude Bajard, Julien Eynard, and Nabil Merkiche. 2016. Multi-fault Attack Detection for RNS Cryptographic Architecture. In *Computer Arithmetic (ARITH), 2016 IEEE 23nd Symposium on.* IEEE, 16–23.

[4] J-C Bajard and Laurent Imbert. 2004. A full RNS implementation of RSA. *IEEE Trans. Comput.* 53, 6 (2004), 769–774.

[5] Ferruccio Barsi and Piero Maestrini. 1974. Error detection and correction by product codes in residue number systems. *IEEE Trans. Comput.* 100, 9 (1974), 915–924.

[6] David T Brown. 1960. Error detecting and correcting binary codes for arithmetic operations. *IRE Transactions on Electronic Computers* 3 (1960), 333–337.

[7] YG.C. Cardarilli and M. Re ; R. Lojacono. 1998. RNS-to-binary conversion for efficient VLSI implementation. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 45 (1998), 667–669.

[8] Chip-Hong Chang, Amir Sabbagh Molahosseini, Azadeh Alsadat Emrani Zarandi, and Tian Fatt Tay. 2015. Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. *IEEE circuits and systems magazine* 15, 4 (2015), 26–44.

[9] Jen-Shiun Chiang and Mi Lu. 1991. Floating-point numbers in residue number systems. *Computers & Mathematics with Applications* 22, 10 (1991), 127–140.

[10] Rooju Chokshi, Krzysztof S Berezowski, Aviral Shrivastava, and Stanislaw J Piestrak. 2009. Exploiting residue number system for power-efficient digital signal processing in embedded processors. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems.* ACM, 19–28.

[11] B. Deng, S. Srikanth, E. R. Hein, P. G. Rabbat, T. M. Conte, E. DeBenedictis, and J. Cook. 2016. Computationally-redundant energy-efficient processing for y'all (CREEPY). In *IEEE International Conference on Rebooting Computing (ICRC).* 1–8.

[12] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (Oct 1974), 256–268.

[13] Elio D Di Claudio, Gianni Orlandi, and Francesco Piazza. 1993. A systolic redundant residue arithmetic error correction circuit. *IEEE Trans. Comput.* 42, 4 (1993), 427–432.

[14] Elio D Di Claudio, Francesco Piazza, and Gianni Orlandi. 1995. Fast combinatorial RNS processors for DSP applications. *IEEE transactions on computers* 44, 5 (1995), 624–633.

[15] Shlomi Dolev, Sergey Frenkel, Dan E Tamir, and Vladimir Sinelnikov. 2013. Preserving Hamming Distance in Arithmetic and Logical Operations. *Journal of Electronic Testing* 29, 6 (2013), 903–907.

[16] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and others. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *Microarchitecture,MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on.* IEEE, 7–18.

[17] M Etzel and W Jenkins. 1980. Redundant residue number systems for error detection and correction in digital filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28, 5 (1980), 538–545.

[18] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. 2009. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 283–296.

[19] Ph Forin. 1989. Vital coded microprocessor principles and application for various transit systems. *IFAC Control, Computers, Communications* (1989), 79–84.

[20] Swaroop Ghosh, Patrick Ndai, and Kaushik Roy. 2008. A novel low overhead fault tolerant Kogge-Stone adder using adaptive clocking. In *Design, Automation and Test in Europe, 2008. DATE'08*. IEEE, 366–371.

[21] Vik Tor Goh and Mohammad Umar Siddiqi. 2008. Multiple error detection and correction based on redundant residue number systems. *IEEE Transactions on Communications* 56, 3 (2008).

[22] Oded Goldreich, Dana Ron, and Madhu Sudan. 1999. Chinese remaindering with errors. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*. ACM, 225–234.

[23] Meeta S Gupta, Krishna K Rangan, Michael D Smith, Gu-Yeon Wei, and David Brooks. 2008. DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors. In *High Performance Computer Architecture, HPCA, IEEE 14th International Symposium on*. IEEE, 381–392.

[24] Nor Zaidi Haron and Said Hamdioui. 2011. Redundant residue number system code for fault-tolerant hybrid memories. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 7, 1 (2011), 4.

[25] C. W. Hastings. 1966. Automatic detection and correction of errors in digital computers using residue arithmetic. In *Region Six Annu. Conf.* IEEE, 429–464.

[26] Yuang-Ming Hsu and EE Swartzlander. 1992. Time redundant error correcting adders and multipliers. In *Defect and Fault Tolerance in VLSI Systems, 1992. Proceedings., 1992 IEEE International Workshop on*. IEEE, 247–256.

[27] Yuang-Ming Hsu and EE Swartzlander. 1992. Time redundant error correcting adders and multipliers. In *Defect and Fault Tolerance in VLSI Systems, Proceedings., International Workshop on*. IEEE, 247–256.

[28] Ching Yu Hung and Behrooz Parhami. 1994. Fast RNS division algorithms for fixed divisors with application to RSA encryption. *Inform. Process. Lett.* 51, 4 (1994), 163–169.

[29] IBM. 2014. IBM Power System E880 server, an IBM POWER8 technology-based system, addresses the requirements of an industry-leading enterprise class system. (2014).

[30] Barry W Johnson, James H Aylor, and Haytham H Hana. 1988. Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder. *IEEE journal of solid-state circuits* 23, 1 (1988), 208–215.

[31] Barry W Johnson, James H Aylor, and Haytham H Hana. 1988. Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder. *journal of solid-state circuits* 23, 1 (1988), 208–215.

[32] Rajendra S. Katti. 1996. A new residue arithmetic error correction scheme. *IEEE transactions on computers* 45, 1 (1996), 13–19.

[33] Osnat Keren, Ilya Levin, Vladimir Ostrovsky, and Beni Abramov. 2008. Arbitrary error detection in combinational circuits by using partitioning. In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*. IEEE, 361–369.

[34] Asif Islam Khan, Korok Chatterjee, Juan Pablo Duarte, Zhongyuan Lu, Angada Sachid, Sourabh Khandelwal, Ramamoorthy Ramesh, Chenming Hu, and Sayeef Salahuddin. 2016. Negative capacitance in short-channel FinFETs externally connected to an epitaxial ferroelectric capacitor. *IEEE Electron Device Letters* 37, 1 (2016), 111–114.

[35] Asif Islam Khan and Sayeef Salahuddin. 2015. 4 Extending CMOS with negative capacitance. *CMOS and Beyond: Logic Switches for Terascale Integrated Circuits* (2015), 56–76.

[36] Asif I Khan, Chun W Yeung, Chenming Hu, and Sayeef Salahuddin. 2011. Ferroelectric negative capacitance MOSFET: Capacitance tuning & antiferroelectric operation. In *Electron Devices Meeting (IEDM), 2011 IEEE International*. IEEE, 11–3.

[37] EV Krekhov, Al-r A Pavlov, AA Pavlov, PA Pavlov, DV Smirnov, AN Tsar'kov, PA Chistopol'skii, AV Shandrikov, BA Sharikov, and DA Yakimov. 2008. A method of monitoring execution of arithmetic operations on computers in computerized monitoring and measuring systems. *Measurement Techniques* 51, 3 (2008), 237–241.

[38] Hari Krishna, Bal Krishna, Kuo-Yu Lin, and Jenn-Dong Sun. 1994. *Computational Number Theory and Digital Signal Processing: Fast Algorithms and Error Control Techniques*. Vol. 6. CRC Press.

[39] Hari Krishna, K-Y Lin, and J-D Sun. 1992. A coding theory approach to error control in redundant residue number systems. I. Theory and single error correction. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39, 1 (1992), 8–17.

[40] Daniel Lipetz and Eric Schwarz. Self Checking in Current Floating-Point Units. In *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic (ARITH '11)*. IEEE Computer Society, Washington, DC, USA, 73–76.

[41] Chao-Kai Liu. 1972. *Error-correcting-codes in computer arithmetic*. Technical Report. DTIC Document.

[42] Hao-Yung Lo and Ting-Wei Lin. 2013. Parallel Algorithms for Residue Scaling and Error Correction in Residue Arithmetic. *Wireless Engineering and Technology* 4, 04 (2013), 198.

[43] Florence Jessie MacWilliams and Neil James Alexander Sloane. 1977. *The theory of error-correcting codes*. Elsevier.

[44] Daniel Marienfeld, Egor S Sogomonyan, Vitalij Ocheretnij, and M Gossel. 2005. New self-checking output-duplicated booth multiplier with high fault coverage for soft errors. In *Test Symposium, 2005. Proceedings. 14th Asian.* IEEE, 76–81.

[45] J Mathew, S Banerjee, P Mahesh, DK Pradhan, AM Jabir, and SP Mohanty. 2010. Multiple bit error detection and correction in GF arithmetic circuits. In *Electronic System Design (ISED), 2010 International Symposium on.* IEEE, 101–106.

[46] A. McMenamin. 2013. The End of Dennard Scaling. (2013).

[47] Elias Mizan, Tileli Amimeur, and Margarida F Jacome. 2007. Self-imposed temporal redundancy: An efficient technique to enhance the reliability of pipelined functional units. In *Computer Architecture and High Performance Computing, SBAC-PAD. 19th International Symposium on.* IEEE, 45–53.

[48] Michael Nicolaidis. 2003. Carry checking/parity prediction adders and ALUs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11, 1 (2003), 121–128.

[49] Michael Nicolaidis and Hakim Bederr. 1994. Efficient implementations of self-checking multiply and divide arrays. In *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.* IEEE, 574–579.

[50] Michael Nicolaidis and RO Duarte. 1998. Design of fault-secure parity-prediction booth multipliers. In *Design, Automation and Test in Europe, 1998., Proceedings.* IEEE, 7–14.

[51] Eric B Olsen. 2015. Introduction of the Residue Number Arithmetic Logic Unit With Brief Computational Complexity Analysis (Rez-9 soft processor). *Whitepaper, Digital System Research* (2015).

[52] Amos Omondi and Benjamin Premkumar. Residue Number Systems: Theory and Implementation. Imperial College Press.

[53] Glenn A. Orton, Lloyd E. Peppard, and Stafford E. Tavares. 1992. New fault tolerant techniques for residue number systems. *IEEE transactions on computers* 41, 11 (1992), 1453–1464.

[54] Janak H. Patel and Leona Y. Fung. 1982. Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Trans. Computers* 31, 7 (1982), 589–595.

[55] Song Peng and Rajit Manohar. 2005. Fault tolerant asynchronous adder through dynamic self-reconfiguration. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on.* IEEE, 171–178.

[56] Song Peng and Rajit Manohar. 2005. Fault tolerant asynchronous adder through dynamic self-reconfiguration. In *Computer Design: VLSI in Computers and Processors, ICCD. Proceedings.International Conference on.* IEEE, 171–178.

[57] AP Preethy and D Radhakrishnan. 1999. A 36-bit balanced moduli MAC architecture. In *Circuits and Systems, 1999. 42nd Midwest Symposium on*, Vol. 1. IEEE, 380–383.

[58] AP Preethy and D Radhakrishnan. 2000. RNS-based logarithmic adder. *IEE Proceedings-Computers and Digital Techniques* 147, 4 (2000), 283–287.

[59] Vijaya Ramachandran. 1983. Single residue error correction in residue number systems. *IEEE transactions on computers* 32, 5 (1983), 504–507.

[60] J Ramirez, A Garcia, S Lopez-Buedo, and A Lloris. 2002. RNS-enabled digital signal processor design. *Electronics Letters* 38, 6 (2002), 266–268.

[61] Thammavarapu RN Rao. 1970. Biresidue error-correcting codes for computer arithmetic. *IEEE Transactions on computers* 100, 5 (1970), 398–402.

[62] Wenjing Rao and Alex Orailoglu. 2008. Towards fault tolerant parallel prefix adders in nanoelectronic systems. In *Design, Automation and Test in Europe, 2008. DATE'08.* IEEE, 360–365.

[63] Wenjing Rao, Alex Orailoglu, and Ramesh Karri. 2006. Fault identification in reconfigurable carry lookahead adders targeting nanoelectronic fabrics. In *Test Symposium, 2006. ETS'06. Eleventh IEEE European.* IEEE, 63–68.

[64] Stefan Rusu. 2010. Multi-Domain Processors Design Overview. *ISCA tutorial on Multi-domain Processors: Challenges, Design Methods, and Recent Developments* (June 2010).

[65] Sayeef Salahuddin and Supriyo Datta. 2008. Can the subthreshold swing in a classical FET be lowered below 60 mV/decade?. In *Electron Devices Meeting, 2008. IEDM 2008. IEEE International.* IEEE, 1–4.

[66] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. 2010. ANB-and ANBDmem-encoding: detecting hardware errors in software. In *International Conference on Computer Safety, Reliability, and Security.* Springer, 169–182.

[67] Avik Sengupta and Balasubramaniam Natarajan. 2013. Performance of systematic RRNS based space-time block codes with probability-aware adaptive demapping. *IEEE Transactions on Wireless Communications* 12, 5 (2013), 2458–2469.

[68] Y. Shimazaki, R. Zlatanovici, and B. Nikolic. 2004. A shared-well dual-supply-voltage 64-bit ALU. *Journal of Solid-State Circuits* 39, 3 (2004), 494–500.

[69] Sriseshan Srikanth, Bobin Deng, and Thomas M Conte. 2016. A Brief Survey of Non-Residue Based Computational Error Correction. *arXiv preprint arXiv:1611.03099* (2016).

[70] S. Srikanth, P. G. Rabbat, E. R. Hein, B. Deng, T. M. Conte, E. DeBenedictis, J. Cook, and M Frank. Memory System Design for Ultra Low Power, Computationally Error Resilient Processor Microarchitectures. In *International Symposium on High Performance Computer Architecture (HPCA),2018.* [to appear].

[71] C-C Su and H-Y Lo. 1990. An algorithm for scaling and single residue error correction in residue number systems. *IEEE Trans. Comput.* 39, 8 (1990), 1053–1064.
[72] J-D Sun and Hari Krishna. 1992. A coding theory approach to error control in redundant residue number systems. II. Multiple Error detection and correction. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39, 1 (1992), 18–34.
[73] J-D Sun, Hari Krishna, and KY Lin. 1992. A superfast algorithm for single-error correction in RRNS and hardware implementation. In *Circuits and Systems, 1992. ISCAS'92. Proceedings., 1992 IEEE International Symposium on*, Vol. 2. IEEE, 795–798.
[74] Yan Sun, Minxuan Zhang, Shaoqing Li, and Yali Zhao. 2010. Cost effective soft error mitigation for parallel adders by exploiting inherent redundancy. In *IC Design and Technology (ICICDT), 2010 IEEE International Conference on*. IEEE, 224–227.
[75] Andraos Sweidan and Ahmad A Hiasat. 2001. On the theory of error control based on moduli with common factors. *Reliable computing* 7, 3 (2001), 209–218.
[76] Nicholas S Szabo and Richard I Tanaka. 1967. *Residue arithmetic and its applications to computer technology.* McGraw-Hill.
[77] J. Tan and O. Rosen. 2004. Process for determining competing cause event probability and/or system availability during the simultaneous occurrence of multiple events. (April 22 2004). https://www.google.com/patents/US20040078167 US Patent App. 10/272,156.
[78] Yangyang Tang, Emmanuel Boutillon, Christophe Jégo, and Michel Jézéquel. 2010. A new single-error correction scheme based on self-diagnosis residue number arithmetic. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*. IEEE, 27–33.
[79] Thian Fatt Tay and Chip-Hong Chang. 2014. A new algorithm for single residue digit error correction in Redundant Residue Number System. In *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*. IEEE, 1748–1751.
[80] Thian Fatt Tay and Chip-Hong Chang. 2016. A non-iterative multiple residue digit error detection and correction algorithm in RRNS. *IEEE transactions on computers* 65, 2 (2016), 396–408.
[81] T. N. Theis. 2012. (keynote) in quest of a fast, low-voltage digital switch. *ECS Transactions* 45, 6 (2012), 3–11.
[82] T. N. Theis and P. M. Solomon. 2010. In Quest of the Next Switch: Prospects for Greatly Reduced Power Dissipation in a Successor to the Silicon Field-Effect Transistor. *Proc. IEEE* 98, 12 (Dec 2010), 2005–2014.
[83] Mojtaba Valinataj and Saeed Safari. 2007. Fault tolerant arithmetic operations with multiple error detection and correction. In *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*. IEEE, 188–196.
[84] Dilip P Vasudevan and Parag K Lala. 2005. A technique for modular design of self-checking carry-select adder. In *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on*. IEEE, 325–333.
[85] Dilip P Vasudevan, Parag K Lala, and James Patrick Parkerson. 2007. Self-checking carry-select adder design based on two-rail encoding. *IEEE Transactions on Circuits and Systems I: Regular Papers* 54, 12 (2007), 2696–2705.
[86] John Von Neumann. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies* 34 (1956), 43–98.
[87] E George Walters III, Mark G Arnold, and Michael J Schulte. 2003. Using truncated multipliers in DCT and IDCT hardware accelerators. In *Optical Science and Technology, SPIE's 48th Annual Meeting*. International Society for Optics and Photonics, 573–584.
[88] Ute Wappler and Christof Fetzer. 2007. Hardware failure virtualization via software encoded processing. In *Industrial Informatics, 2007 5th IEEE International Conference on*, Vol. 2. IEEE, 977–982.
[89] R. W. Watson. 1965. Error detection and correction and other residue interacting operations in a residue redundant number system. Univ. California, Berkeley.
[90] Richard W Watson and Charles W Hastings. 1966. Self-checked computation using residue arithmetic. *Proc. IEEE* 54, 12 (1966), 1920–1931.
[91] Hanshen Xiao, Hari Krishna Garg, Jianhao Hu, and Guoqiang Xiao. 2016. New Error Control Algorithms for Residue Number System Codes. *ETRI Journal* 38, 2 (2016), 326–336.
[92] Li Xiao and Xiang-Gen Xia. 2015. Error correction in polynomial remainder codes with non-pairwise coprime moduli and robust Chinese remainder theorem for polynomials. *IEEE Transactions on Communications* 63, 3 (2015), 605–616.
[93] SS-S Yau and Yu-Cheng Liu. 1973. Error correction in redundant residue number systems. *IEEE Trans. Comput.* 100, 1 (1973), 5–11.
[94] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. 2003. RSA speedup with Chinese remainder theorem immune against hardware fault cryptanalysis. *IEEE Transactions on computers* 52, 4 (2003), 461–472.
[95] Pengsheng Yin and Lei Li. 2013. A new algorithm for single error correction in RRNS. In *Communications, Circuits and Systems (ICCCAS), 2013 International Conference on*, Vol. 2. IEEE, 178–181.

Chapter 5

Memory Addressing for an RRNS-based Architecture

This paper is to appear in the proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA).

Memory System Design for Ultra Low Power, Computationally Error Resilient Processor Microarchitectures

Sriseshan Srikanth*, Paul G. Rabbat†, Eric R. Hein*, Bobin Deng*, Thomas M. Conte*,
Erik DeBenedictis‡, Jeanine Cook‡ and Michael P. Frank‡
*Georgia Institute of Technology. Email: {seshan}{ehein6}{bdeng}{tom}@gatech.edu
†Intel Corporation. Email: paul.g.rabbat@intel.com
‡Sandia National Laboratories. Email: {epdeben}{jeacook}{mpfrank}@sandia.gov

***Abstract*—Dennard scaling ended a decade ago. Energy reduction by lowering supply voltage has been limited because of guard bands and a subthreshold slope of over 60mV/decade in MOSFETs. On the other hand, newly-proposed logic devices maintain a high on/off ratio for drain currents even at significantly lower operating voltages. However, such ultra low power technology would eventually suffer from intermittent errors in logic as a result of operating close to the thermal noise floor. Computational error correction mitigates this issue by efficiently correcting stochastic bit errors that may occur in computational logic operating at low signal energies, thereby allowing for energy reduction by lowering supply voltage to tens of millivolts.**

Cores based on a Redundant Residual Number System (RRNS), which represents a number using a tuple of smaller numbers, are a promising candidate for implementing energy-efficient computational error correction. However, prior RRNS core microarchitectures abstract away the memory hierarchy and do not consider the power-performance impact of RNS-based memory addressing. When compared with a non-error-correcting core addressing memory in binary, naive RNS-based memory addressing schemes cause a slowdown of over 3x/2x for inorder/out-of-order cores respectively. In this paper, we analyze RNS-based memory access pattern behavior and provide solutions in the form of novel schemes and the resulting design space exploration, thereby, extending and enabling a tangible, ultra low power RRNS based architecture.

I. Introduction

We hit the power wall in 2005, which meant that increasing clock frequency was no longer a technique to improve performance. Single core performance plateaued as a result of the power wall although there is more instruction level parallelism (ILP) inherent in programs, waiting to be exploited [11], [35], [58]. The community has since then adopted multiple cores and specialized accelerators to make better use of Moore's law, albeit at the cost of programmability. The phenomenon of transistors retaining their power density as they scaled down (known as Dennard Scaling [22]) ended a decade ago [69], meaning that adding more and more transistors no longer improved performance without also significantly increasing energy consumption. A sure-shot mitigation technique is to strive to improve the fundamental single core power-performance profile from the ground up.

Dynamic power scales proportionately with frequency and with the square of operating voltage, therefore, lowering V_{dd} is more beneficial. Although the theoretical lower limit for V_{dd} for inverter functionality is $2\frac{kT}{q}$ (or about $36mV$) [49], [96], [110], there are challenges to operating at this level [13]. With conventional MOSFETs, reducing supply voltage below ˜0.7V results in increased switching delay, irrespective of channel length. Coupled with their gentle subthreshold slope ($> 60mV/decade$), the upshot is that V_{dd} reduction actually causes higher leakage energy, negating the benefits of the reduction in the first place and furthermore forcing a significantly slower clock rate.

Theis and Solomon [105] suggest that new device concepts [76] within the purview of two-dimensional lithography technology, such as tunneling FETs, enable reduction of the $\frac{1}{2}CV^2$ energy to small multiples of kT, without resulting in a significantly low switching speed [104] when compared to MOSFETs. Similarly, research on ferroelectric transistors, *aka* negative capacitance FETs (NCFETs) demonstrates a sub-$60mV/dec$ slope as well as a higher drive current [50]–[52], [87], both of which are necessary in rendering V_{dd} reduction beneficial to energy reduction without significantly sacrificing performance, when compared to MOSFETs.

These next generation devices are fast switching even at few tens of millivolts, but as a result, are vulnerable to thermal noise perturbations. This translates into *intermittent, stochastic bit errors in logic*. With signal energies approaching the kT noise floor, future architectures will need to treat reliability as a first class citizen, by employing efficient computational error correction.

A. Computational Error Correction

Standard error correcting codes (ECC) [66] have already been adopted into modern memory systems. These codes accommodate errors occurring in storage and communication/network traffic, but are not able to protect computational logic. The naive approach to computational error correction is triple modular redundancy (TMR) [109], requiring over a 200% overhead in area and energy for single error correcting/double error detecting (SECDED) capability. Several techniques in the form of arithmetic codes such as AN codes [9], [28], [29], [62], [88], [111], self-checking [45], [48], [67], [73]–[75], [107] and self-correcting [25], [32], [41], [55], [68], [79], [83], [84], [95], [106] adders and multipliers have since been devised. Orthogonally, there have been proposals that employ redundancy at a higher granularity, such as timing speculation (wherein error correction capability is limited to circuit timing violations) [26], [37], partial pipeline replication [1] or checkpoint-rollback-recovery such as those in IBM POWER6/7/8 and z10/196 [8], [16], [44], [61] processors, and various Intel Corporation [90] and Sun Microsystems mainframes [43]. While these are more efficient than naive TMR, they come with limitations on their error model, their area overheads are still over 100% and/or they incur a significant performance penalty [91] (e.g., owing to the fact that they leverage temporal redundancy in an effort to minimize area overhead).

There exists a class of computationally error resilient codes based on the residue number system (RNS) [31], that

This work was approved for public release by Sandia National Laboratories SAND2017-12733.

Table I: A (4, 2)-RRNS example with the simplified base set (3, 5, 2, 7, 11, 13). % is the mod operator. Range is 210, with 11 and 13 being the redundant bases. (Reproduced from [21].)

Decimal	% 3	% 5	% 2	% 7	% 11	% 13
13	1	3	1	6	2	0
14	2	4	0	0	3	1
13+14=27	(1+2)% 3=0	2	1	6	5	1
	All columns (residues) function independently of one another.					
	An error in any one of these columns (residues) can be corrected by the remaining columns.					

are generally superior to the above techniques in terms of area, energy and latency overheads. The premise of RNS is that a number can be uniquely represented as a tuple of residues, where the residues are the remainder when the number is divided by a set of coprime bases/moduli. Each residue itself may be represented in a weighted radix representation in binary. An example is shown in Table I. Assume the set of bases/moduli to be (3, 5, 2, 7); then, the number 13 can be uniquely mapped to the tuple (1, 3, 1, 6) as a result of the Chinese remainder theorem. Observe that (13 mod 3 = 1), (13 mod 5 = 3) and so on. Now, suppose we wish to add 13 (1, 3, 1, 6) to the number 14 (2, 4, 0, 0); this can be achieved by simply (modulo) adding the residues respectively to obtain 27 (0, 2, 1, 6). Observe that ((1 + 2) mod 3 = 0), ((3 + 4) mod 5 = 2) and so on. Critically, the computation occurs with no carries or interaction between residues.

RNS is similarly closed under subtraction and multiplication operations as well, and they too can operate without interaction between residues. This important property has several useful implications:

- As the residues operate with no carries between them, they can operate in parallel. Furthermore, each residue operation's bit width is a fraction of that of the original operation. This translates into improved computational efficiency, which has been proven to be especially useful in digital signal processing (DSP) [19], [24], [81].
- A very large number can be losslessly represented and operated on via many smaller numbers (residues) in parallel. This property is used by the cryptography (RSA) community [3], [42], [116].
- Any bit error caused by a faulty computational logic element is guaranteed to be localized to within the corresponding residue, without impacting any of the other residues.

The last implication is of particular interest because it allows for a robust, efficient method for computational error correction. When redundant bases/moduli are introduced (Redundant RNS or RRNS) [112], the resulting redundant residues form an error correcting code that transforms itself automatically upon arithmetic operations, rather than having to be recomputed afterwards. If a corrupt residue results during an arithmetic operation, it is possible to infer its value using the remaining correct residues in the result. To continue our running example in Table I, the number 13, being less than the product of the initial set of bases/moduli (3, 5, 2, 7), (i.e, 210), it can be uniquely represented using the 4-tuple (1, 3, 1, 6), via the Chinese remainder theorem. As such, we refer to these bases/moduli and residues as non-redundant. Upon adding two redundant bases/moduli, say (11, 13), the resultant redundant residues are therefore (2, 0). These redundant residues, by definition, are not necessary for representation, but provide a way to recover from errors. Given the (4, 2)-tuples for 13 (1, 3, 1, 6, 2,0) and 14 (2, 4, 0, 0, 3,1), a storage/transmission/computation error arising in any one of the residues of 13, 14 or their arithmetic manipulation result, can be corrected using the remaining residues. The running example is summarized in Table I.

The range for an RRNS representation is the product of its non-redundant moduli. Therefore, by choosing a non-redundant base/modulus set to be (199, 233, 194, 239) it becomes possible to represent a 32-bit integer in general in an RNS format, and extending it with a redundant set of (251, 509) allows for an RRNS representation. Such a (4, 2)-RRNS has the following advantages, over and above what RNS provides to us:

- Computational error correction can be achieved with a little over 50% area overhead.
- The SECDED granularity is that of a residue; meaning that such an RRNS is capable of *correcting multi-bit errors as long as they occur within a single residue*, or alternately, is capable of *detecting multi-bit errors as long as they occur within at most 2 residues*.
- Being closed under arithmetic, the correct value is *preserved* across a chain of dependent operations. Therefore, it is not necessary to incur the overhead of an RRNS error correction/detection after every operation.
- Due to the above, RRNS lends us robust computational error correction with relatively insignificant performance penalty, as also evidenced by Deng et al. [21].

Furthermore, there has been a significant body of research on RRNS that strives to make it more efficient algorithmically for error resilience [5], [6], [17], [23], [27], [33], [34], [38], [47], [56], [57], [63], [77], [80], [82], [89], [92]–[94], [97], [98], [101]–[103], [113]–[115], [117] and division [4], [30], [39], [40], [42], [63], [64], [92], [99]. Although typically limited to detection, residue based logic protection (including that for floating point units and vectorized units) has widespread used in several commercial high-end server processors [8], [16], [43], [61], [90]. Clearly, the RRNS approach of computational error correction is ranked among the highest in terms of error correction capability and efficiency.

An efficient RRNS-based architecture is a viable compiler target for contemporary general-purpose as well as scientific computing applications, *in addition to being able to work efficiently with novel devices that operate close to the kT noise floor*. We strongly believe that single-thread processor performance scaling that has been stalled since the mid-2000s can be restarted via RRNS.

B. The Problem

While RRNS is clearly attractive for designing ultra-low-power, computationally error resilient microarchitectures, prior work on RRNS has abstracted away the memory hierarchy for simplicity. Thus, an RRNS microarchitecture hits a performance bottleneck when connected to non-ideal, real world memory systems. Memory is addressed in binary in conventional processors, whereas an RRNS compute core natively generates memory addresses as a tuple of residues. There are two naive approaches to this memory interface problem:

Binary. Convert the tuple-of-residues format to binary and address memory in binary as usual. This approach imposes a severe latency and energy penalty on every instruction fetch, load and store operation. This overhead is due to the multi-step nature of each RNS to binary conversion, which, nominally costs 8 cycles in the form of add and table lookup operations. According to our results, this slowdown is 3× on average for in-order cores and 2× for out-of-order (OoO) cores.

Rns_concat. Another straightforward, although naive approach is to concatenate the native tuple-of-residues and use the result as the memory address. Unfortunately, this technique destroys spatial locality of sequential memory accesses, rendering caches largely ineffective and causing application slowdowns of over 3× on average for in-order cores and 4× for OoO cores.

There are other overheads associated with RRNS, such as non-trivial comparison and boolean operations. However, the overheads due to memory addressing inefficiencies are significantly higher. The memory hierarchy is accessed for *each* instruction (PC) in addition to memory instructions (LD/ST). In contrast, comparison and boolean op instructions are less frequent, even if a targeted code generator does not minimize such ops. Furthermore, even with a naive implementation, the penalty for such ops is less than that of a cache miss. In an independent study (that ignores memory addressing inefficiencies), we found that the impact of slow comparison, boolean operations as well as consistency checking operations due to RRNS makes programs run just about 20% slower than a traditional core. Yet, due to RRNS, we realize significant energy savings, rendering energy-delay-product benefits of about 2× when compared to traditional non-error-correcting cores, in spite of these overheads.

In spite of its strong potential to restart single thread performance scaling, an RRNS processor is not competitive with traditional designs due to memory addressing inefficiencies outlined above. The energy savings would be overshadowed by the decrease in memory system performance. This paper is the first study of its kind to our knowledge to focus on the RRNS memory access problem.

C. Contributions

This paper makes the following contributions:

1) Extends an RRNS microarchitecture - one that is capable of reliably functioning with ultra-low energy logic devices that operate near the kT thermal noise floor at tens of millivolts - to support efficient memory hierarchy access.
2) Proposes and analyzes an efficient, novel translation scheme (*rns_sub*) with locality properties similar to that of *binary*, but with a fraction of its performance/energy overhead.
3) Reduces the performance impact of the naive *binary* approach by using a TLB-like structure called the *Conversion Lookaside Buffer* (CLB) to cache conversions.
4) Proposes a technique to improve spatial locality in the native, zero-overhead translation scheme (*rns_concat*) via a hybrid compiler approach and a modified programming model.
5) Constructs a design space from the schemes proposed and from the analysis of the resultant memory access pattern behavior, along with a detailed cost-benefit analysis.

II. RRNS Core Microarchitecture

Having already presented an overview of the general workings of RNS and RRNS, we now provide a formal description for completeness (proofs omitted for brevity). We then describe an overview of a generic RRNS compute core. Readers who are not interested in the formality can safely skip ahead to Section II-C.

A. Residue Number System (RNS)

Let $B = \{m_i \in \mathbb{N} \ for \ i = 1,2,3,...,n\}$ be a set of n co-prime natural numbers, which we shall refer to as bases or moduli. $M = \prod_{i=1}^{n} m_i$ defines the range of natural numbers that can be bijectively represented by an RNS system that is defined by the set of bases B. Specifically, for x such that $x \in \mathbb{N}, \ x < M$, then, x can be represented as the following tuple: $(|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n})$, where $|x|_m = x \ mod \ m$. Each term in this n-tuple is referred to as a residue. If necessary the value of x can be regenerated from the tuple of residues using a series of addition and table lookup operations via the Chinese remainder theorem, mixed-radix conversion, or macro-coefficient extraction.

Addition, subtraction and multiplication are closed under RNS, rendering the residues to be mutually independent *wrt* arithmetic. In other words, given $x,y \in \mathbb{N}, \ x,y < M$, we have $|x \ op \ y|_m = ||x|_m \ op \ |y|_m|_m$, where op is any add/subtract/multiply operation.

B. Redundant RNS

To augment RNS with fault tolerance, r redundant bases are introduced. The set of moduli now contains n non-redundant and r redundant moduli: $B = \{m_i \in \mathbb{N}$ for $i = 1,2,3,...,n,n+1,...,n+r\}$. The reason these extra bases are redundant is because any natural number smaller than $M \ (= \prod_{i=1}^{n} m_i)$ can still be represented uniquely by its n non-redundant residues. Recall that the introduction of r redundant residues renders a computationally resilient *error code* because of the fact that all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N}, \ x < M$, its RRNS representation is as follows: $(|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n}, |x|_{m_{n+1}}, ..., |x|_{m_{n+r}})$, i.e., containing n non-redundant residues as well as r redundant residues.

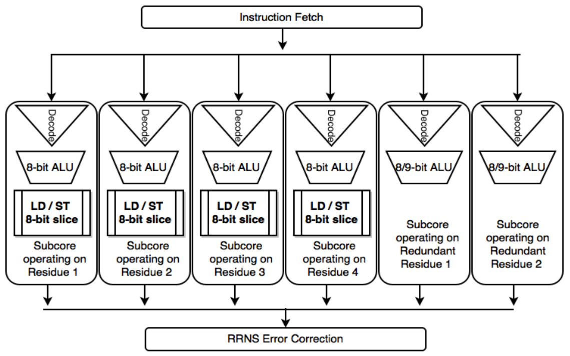
We refer the interested reader to [112] for details of the multi-bit error correction operation, which involves addition, multiplication and table lookup operations. This correction capability increases with r, tolerating upto $\frac{r}{2}$ errant residues [33]. There are proposals to perform fractional multiplication [112] and to represent floating point numbers [18] using RNS. The key idea to extend this to RRNS is to protect the exponent and mantissa separately, as they transform differently upon arithmetic operations. A detailed treatment of these concepts is beyond the scope of this paper.

For fault tolerance to work, certain conditions must be satisfied by B [112]. Given these, published work suggests that the most efficient conversion to *binary* algorithm (for $n = 4$, independent of r) nominally costs 8 cycles and is possible via macro-coefficient extraction [112]. More efficient conversion algorithms for RNS exist, such as those that use Mersenne primes (or other specific structural properties) as bases [14], [15], [71], but no known RRNS algorithms exist.

In this paper, $(n,r) = (4,2)$ and moduli set used is (199, 233, 194, 239, 251, 509), whose non-redundant representable range is roughly that of a 32 bit unsigned integer. A different set of bases may also be used for reasons such as a higher representable range, more efficient arithmetic, improved reliability characteristics etc., the details of which are beyond the scope of this paper. We assume the aforementioned bases in our evaluation but are careful not to over-fit our architectural suggestions and insights to this specific set.

C. Anatomy of an RRNS Core Microarchitecture

Figure 1 depicts a generic schematic of an RRNS core with 4 non-redundant *sub*-cores and 2 redundant ones. This schematic is similar to prior RRNS core microarchitecture proposals [21], [112]. Each *sub*-core is associated with exactly one base modulus and thereby operates on its own



Instruction Fetch
Decode
8-bit ALU
8/9-bit ALU
LD / ST 8-bit slice
Subcore operating on Residue 1
Subcore operating on Residue 2
Subcore operating on Residue 3
Subcore operating on Residue 4
Subcore operating on Redundant Residue 1
Subcore operating on Redundant Residue 2
RRNS Error Correction

Figure 1: Generic high level schematic of an RRNS error correcting compute core with 4 non-redundant *sub*-cores and 2 redundant *sub*-cores.

residue, with the register file and highest level data cache being distributed into the 6 subcores on a per-residue basis.

Error model. Recall that such a core operates entirely on RRNS data and literals, meaning that there are no unnecessary (and expensive) conversions to and from *binary*. Therefore, all memory addresses (PC, LD/ST) are in RRNS form, including any pointer arithmetic. Another upshot of operating entirely on RRNS data is that control-path errors manifest themselves as data errors, meaning that they can be handled simply by handling the data error. For example, if there is an error in bypass logic in a subcore, or, if a faulty decoder in one of the subcores causes it to perform a multiplication instead of an addition, the resultant residue for that subcore would have an erroneous value, but can be recovered from the remaining 5 residues that were a result of the correct addition operation. Finally, as instructions themselves don't undergo modification, ECC is sufficient to protect them [21]. The architecture proposed in [21] is able to ensure reliable operation as long as at most one subcore is in error (possibly multi-bit) between two error correction operations. Circuit-hardening or using high V_{dd} for the error correction logic is proposed to ensure its reliable operation. Because of latching effects, SRAM transistors have different (more prone to errors) reliability characteristics when compared to logic, and as such, are modeled with 100× the error probability of logic transistors.

Overheads. In terms of area, such an RRNS error correcting core requires 2× the area of a traditional non-error-resilient core. However, a large fraction of this overhead is from LUTs necessary in the error correction operation; a (4, 1)-RRNS core that simply detects errors is in fact 34% smaller in area when compared to a traditional core. In an independent study that implements the error model above (but ignores memory addressing inefficiencies), we found that the overhead due to slow comparison operations, boolean operations as well as RRNS error correction operations resulted in an overhead of just about 20% in runtime when compared to a traditional, non-error-correcting core over general purpose (SPEC2006) as well as computationally intensive workloads (such as FFT, matrix multiplication etc.). However, RRNS enables lowering of signal energy to few tens of millivolts, and results in about a 2× improvement in energy-delay-product, in spite of these overheads. The engineering details of this study are beyond the scope of this paper and we omit them for space constraints.

Abstractions for the memory interface. The presence of a Load/Store unit in the redundant *sub*-cores is implementation dependent. For the purposes of this paper, we assume that the core is responsible for generating *valid* memory requests in the Memory Address Register (MAR). This is possible by either (a) inserting an RRNS consistency check before a memory access, or (b) by enabling a checkpointing mechanism for rollback/recovery in case a memory request to an illegal location is generated. For the latter, a segmentation scheme can be used to flag "wrongly" computed addresses as illegal, thereby triggering checkpoint recovery. As we show in Section III-B, minor perturbations in the native *rns_concat* representation of a memory address causes its value to fluctuate wildly, thereby allowing for such segmentation to work. Note that the consistency check of (a) or the segment check of (b) can be done in parallel with a memory access, and are therefore not on the critical path of the program. Therefore, we omit the Load/Store units in the redundant subcores. Finally, the addressing logic in the memory hierarchy (such as a decoder or an address translator) is assumed to either utilize devices that do not stochastically flip due to thermal noise, or employ self checking logic [36], [86]. Relaxing either assumption reveals a set of implementation dependent tradeoffs and are left for future work. *Without loss of generality, we assume for the purposes of this paper that no explicit error correction is required for such addressing logic in the memory hierarchy*. It follows that the redundant residues can be dropped from the MAR, and that a 32-bit address can be logically represented as a 4-tuple RNS number.

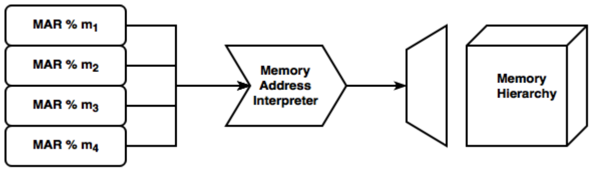


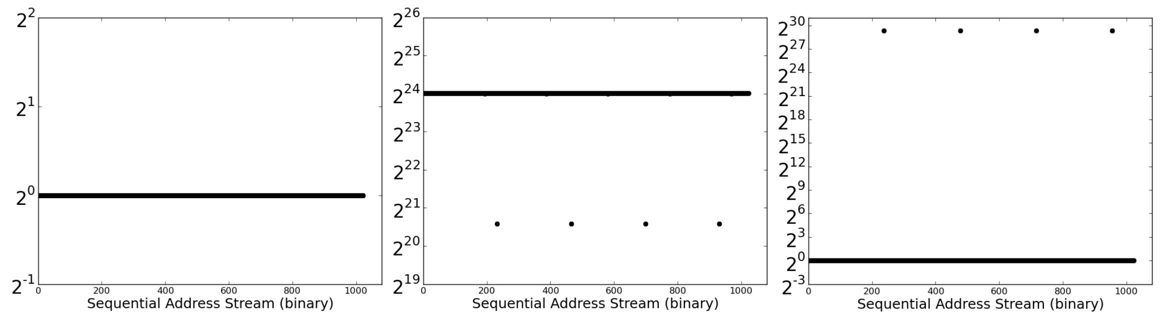
Figure 2: An (R)RNS compute core natively generates memory addresses in the 4-residue tuple format. This is depicted by MAR % m_i in the figure, where m_i is the i^{th} base modulus, % is the modulo operator, and MAR could be one of Program Counter (PC), Load or Store address.

III. Memory Addressing Schemes

As noted in Section II-C, the memory address generated by an RRNS compute core in its native form is in a tuple-of-residues format, where the address itself may be to instructions or data. As depicted in Figure 2, such a 4-tuple must be properly interpreted before the memory hierarchy can be accessed. In this section, we present a basis set of possible *interpretations* of an address.

A. Interpretation Schemes

Binary. The approach assumed by prior work was to convert the 4-tuple of residues to its binary representation and use this as a memory address. From this point on, the system can access the memory hierarchy as conventional computers do. However, this conversion from RNS to *binary* doesn't come for free, and the cost must be paid for each memory access, both cache hits and cache misses. This overhead is a multi-cycle access time increase and an associated increase in energy consumption. This is because conversion from RNS to *binary* is non-trivial: it is a multi-step operation, involving addition and table-lookup operations, and costs 8 CPU cycles nominally. Our experiments indicate that applications suffer from a slowdown of 3× on average for inorder cores and 2× for OoO cores upon using such a naive conversion approach.



(a) Delta between consecutive addresses: *binary* scheme. **(b)** Delta between consecutive addresses: *rns_concat* scheme. **(c)** Delta between consecutive addresses: *rns_sub* scheme.

Figure 3: Spatial locality analysis via visual comparison of the 3 addressing schemes for a sequential stream of addresses by computing the *deltas* of *interpreted* addresses of consecutive addresses of the sequential stream.

$$\{r_1, r_2, r_3, r_4\} \implies binary(\{r_1, r_2, r_3, r_4\})$$

This conversion typically is not a one-time cost as addresses may repeat themselves (locality). As a solution, we propose to cache the conversion itself in a Conversion Lookaside Buffer (**CLB**), in a manner similar to how a TLB caches translations.

Rns_concat. On the other extreme, we have a lightweight interpreter that merely concatenates the 4-tuple, with the resulting 32bit number being treated as the address to the memory hierarchy. More concretely, the 4-residue tuple $\{r_1, r_2, r_3, r_4\}$ is treated as the bitstream $r_1 r_2 r_3 r_4$.

$$\{r_1, r_2, r_3, r_4\} \implies r_1 r_2 r_3 r_4$$

Rns_sub. We define another scheme similar to *rns_concat*, but the lowest residue is subtracted from the 3 other residues in an attempt to preserve locality.

$$\{r_1, r_2, r_3, r_4\} \implies rns\_concat((r_1 - r_4)\%m_1,\\ (r_2 - r_4)\%m_2,\ (r_3 - r_4)\%m_3,\ r_4)$$

where, the RNS base moduli are given by m_i. This scheme is significantly less expensive than *binary* and is slightly more expensive than *rns_concat*.

B. Sequential Address Analysis Example

Figure 3 shows a comparison of how the 3 addressing schemes discussed so far, *binary*, *rns_concat* and *rns_sub*, remap a stream of sequential accesses into the memory address space. On the X-axis is the input value to the Memory Address Interpreter; on the Y-axis is the difference (*delta*) between two consecutive *interpreted* memory addresses.

First, notice that the *delta* for *binary* is always exactly 1; converting the tuple-of-residues back to the binary number they represent results in sequential memory addresses. However, *rns_concat* remaps accesses according to the following function: if the address $X \equiv \{r_1, r_2, r_3, r_4\}$, then $X+1 \equiv \{r_1+1, r_2+1, r_3+1, r_4+1\}$, thereby resulting in a *delta* of 0x01010101 as each residue is 8-bits wide. This is the *delta* value that is seen in the figure as a straight horizontal line slightly above 2^{24}. However, each time a residue overflows, the above constancy claim for *delta* breaks down, generating the discontinuities shown in the plot.

Non-unit delta values will cause consecutive accesses to touch *different* cache lines and memory pages, destroying spatial locality in the access stream. To mitigate this constant *delta* offset seen in *rns_concat*, we define *rns_sub* in an effort to preserve locality. Applying *rns_sub* to X and 1, we observe the following pattern (ignoring modulo overflows):

$$X \equiv \{r_1 - r_4, r_2 - r_4, r_3 - r_4, r_4\}; \quad 1 \equiv \{0,0,0,1\}$$
$$\implies X+1 \equiv \{r_1 - r_4, r_2 - r_4, r_3 - r_4, r_4 + 1\}$$

Therefore, *rns_sub* yields a *delta* value of exactly 1 in the common case, similar to the *binary* scenario. In general, the *rns_sub* representation of any number $n < m_4$ is $\{0,0,0,n\}$, meaning that difference between $X+n$ and X in the *rns_sub* representation is exactly n in the common case. This means that *rns_sub* is capable of preserving the spatial locality that cache and DRAM memory systems need to deliver performance.

While the detailed evaluation of a prefetcher is beyond the scope of this paper, intuitively, prefetchers that work well with *binary* would work well with *rns_sub* as well. With *rns_concat*, however, a stride of 1 must be re-interpreted as a stride of 0x01010101, which is the *delta* between consecutive addresses, significantly altering the operation of most prefetchers.

Although our example analysis in this section is based upon a sequential address stream, the insights are applicable to arbitrary streams that exhibit spatial locality, as is demonstrated via simulation results detailed below.

IV. Design Tradeoffs

Utilizing a basis set of address interpretation schemes outlined in Section III, we now construct and analyze a design space consisting of memory access granularity, Memory Address Interpreter (MAI) configuration and DRAM address interleaving dimensions.

A. Memory Access Granularity

As discussed in Section III, with *rns_concat* based schemes, consecutive addresses do not map onto contiguous memory locations. This causes spatial locality in caches to suffer, prompting us to salvage sequential locality by increasing the memory access granularity to that of a cache line. Under this paradigm, each memory access now refers to a 64-byte chunk of memory rather than a single byte. Clearly, this requires support from the software stack, and we propose an ISA extension, which we name as the SELECT instruction, to help.

The SELECT instruction takes as arguments a base address (represented as *rns_concat*), and a separate offset represented in binary to perform a word lookup within a cache line. This hybrid binary-*rns_concat* representation renders the best of binary on one hand, i.e., sequential locality as intended by the programmer, and that of *rns_concat* on the other hand, i.e., no runtime conversion overhead. A detailed example of how such a compiler transformation may be effected is presented in Section VIII for a word size of 8 bytes, although other word sizes can also be supported.

Representing a portion of the address in binary may seem counter-productive to the reliability of the system, but note that this offset is static (set at compile time) and therefore does not warrant computational error correction, meaning that the ECC protection that comes naturally to instructions (Section II-C) is sufficient to protect this offset as well.

Introducing such a representation comes with a set of limitations in software. First, if each cache line access is to be accompanied by a SELECT instruction, a static code bloat of about 15% occurs. In practice, we expect this bloat to decrease significantly due to spatial locality; once a cache line is loaded, multiple SELECTs can be performed without re-issuing the cache line access. Static code bloat can also be reduced by designing compiler optimizations that further leverage spatial/temporal locality in instruction layout and scheduling. Second, an increased memory access granularity renders arbitrary pointer arithmetic and branch targets tricky to disambiguate at compile time, unless they become aligned to cache-line boundaries. Finally, for general purpose system integration, such a hybrid representation may be required to be extended into the software runtime to avoid an explosion of pages. Alternately, TLB and paging performance may also be improved by employing super pages [72], [100], especially in emerging storage class memories [10] (with segmentation support for security).

The SELECT instruction requires tight integration with the software stack, the detailed implementation of which is beyond the scope of this paper.

B. Memory Address Interpreter (MAI)

Figure 2 presents a simplified view of the memory system. With typical memory systems comprising of L1, L2, L3 caches and a DRAM, the Memory Address Interpreter does not necessarily have to be a singleton unit at the highest level of cache. With a non-inclusive cache hierarchy, the following exhaustive set of legal Memory Address Interpreter configurations are explored:

1) **Ideal**

IBIN This is the ideal configuration where conversion to binary is without any overhead, or equivalently, a non-error-correcting core that uses a conventional weighted binary representation.

2) **Naive**

NL1 This is a naive conversion approach where every memory access is converted to binary before accessing the L1 cache, thereby incurring a conversion overhead of 8 cycles and its associated energy consumption (adders and lookup tables).

NRNS This is a naive conversion approach where the tuple of residues in the MAR is simply concatenated (*rns_concat*).

NMEM This is a naive conversion approach where *rns_concat* is used through the cache hierarchy and a conversion to binary is effected at the memory controller, thereby incurring an overhead of 24 core cycles (this is because the clock domain of the memory controller is nominally about three times slower than that of the core).

3) **Compiler.** These approaches require compiler support:

CRNS This uses compiler support to realize a memory access granularity of 64 bytes (cache line) and simply concatenates the tuple of address residues before accessing L1. In other words, this is **NRNS** when the SELECT instruction is used.

CX **X**$\in${**L2**, **L3**, **MEM**}. These are similar to **CRNS**, except that a conversion to binary is effected before accessing the L2/L3/main memory respectively.

4) **Rns_sub**

SUB This employs the single cycle overhead conversion of *rns_sub* before the L1 cache is accessed.

SUBM This is similar to **SUB** except that a binary conversion is effected before a main memory access.

5) **CLB**

CLBX **X**$\in${**L1_N**, **MEM_N**}. This is similar to **NL1** or **NMEM**, except that an **N**-entry CLB is used in an attempt to hide the performance overhead of the conversion to binary, although at a potentially higher energy cost. Upon a CLB hit, a single cycle conversion is rendered, however, a CLB miss renders an access time equal to that of a binary conversion as usual. Furthermore, **CCLBMEM_N** is similar to **CMEM** but with an **N**-entry CLB at the memory controller.

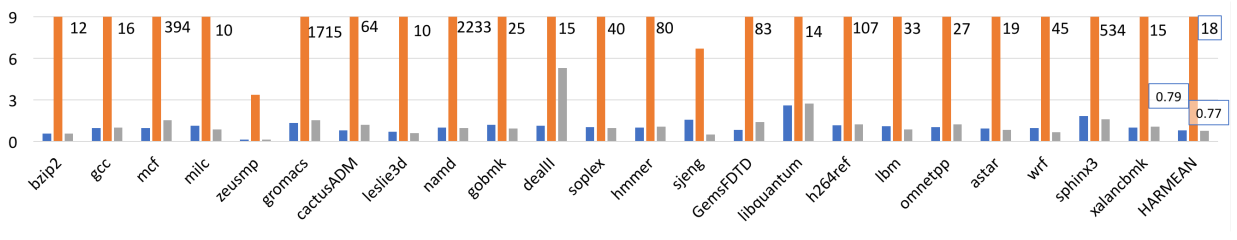
CLBY **Y** is of the form **S_N**. This is similar to **SUBM**, except that the conversion to binary at the memory controller is augmented with an **N**-entry CLB.

Space of valid configurations. Those listed above are deemed representative of an exhaustive brute force sweep. For example, it doesn't make sense to perform a conversion to RNS once a binary conversion or an *rns_sub* has already been effected. Also, mixing *rns_sub* at L1 and conversion to binary at L2/L3 imposes constraints on possible cache configurations[1], hence, we exclude them from our evaluation (although the careful reader may reason about these tradeoffs from the analysis and evaluation presented in this paper).

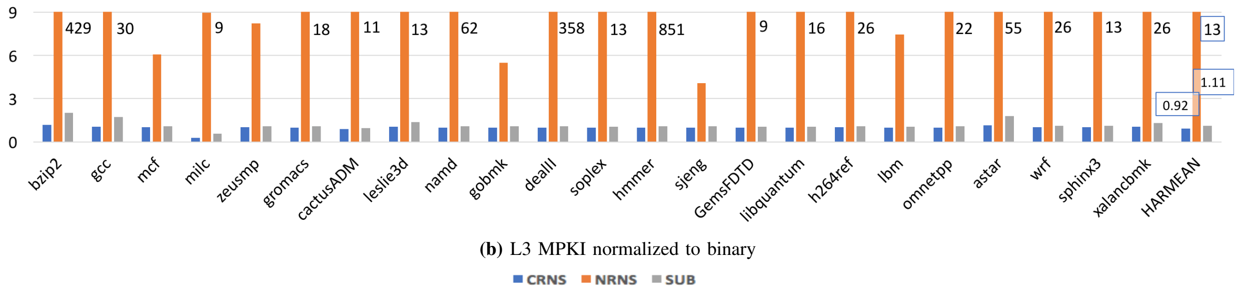
Cache coherence. Cache coherence state is typically maintained at the granularity of a cache line. With the exception of **NRNS**, **NMEM** and **CLBMEM_N**, all the other configurations proposed preserve locality within a cache line at the very least, when compared to **IBIN**. To elaborate, **Compiler** approaches are specifically designed to preserve sequential locality within a cache line. **NL1** and **CLB** based approaches that effectively convert an address to binary prior to L1 trivially preserve locality within a cache line. **Rns_sub** based approaches preserve locality for m_4 consecutive bytes in general (Section III-B); since all the moduli (and m_4, in particular) are greater than 63, therefore, **Rns_sub** preserves cache line locality as well. We conclude that all efficient and well performing configurations proposed are also amenable to traditional, unmodified cache coherence protocols.

Summary. Intuitively, we first observe that the **Compiler** approaches would benefit from the best of locality-preserving properties of *binary* and the no-overhead nature of *rns_concat*, however, at the cost of requiring changes to and tight integration with the software stack. Next, **Rns_sub** approaches would benefit from both its locality-preserving, as well as memory level parallelism inducing properties, at low cost. Finally, **CLB** based approaches would mimic

[1]For example, given the cache configuration in our evaluation, the addresses X (0x4fab84d8) and Y (0x4fab84fc) when subject to *rns_sub* at L1 and conversion to binary at L2, map onto the same L1 index (36) but different L2 indices (165, 166). This makes enforcing a generic non-inclusive property unnecessarily complex.



(a) L1 MPKI normalized to binary



(b) L3 MPKI normalized to binary

Figure 4: Misses Per Kilo Instructions (MPKI – lower is better) of representative RNS based schemes, normalized to a binary scheme. **SUB** successfully retains spatial locality present in the lower order bits of **IBIN**. **CRNS** salvages sequential locality lost by **NRNS**, with the help of compiler support.

IBIN, however, at a significant energy overhead. We will demonstrate in detail in Section V that the relative performance is **Ideal** > **Compiler** > **CLB** > **Rns_sub** >> **Naive**, with **Rns_sub** rendering the most energy efficient architecture along with the added advantage of requiring no support from the software stack.

C. DRAM Address Interleaving

The MAI of Section IV-B directly impacts DRAM address interleaving. While an industry standard memory controller policy is typically undisclosed, we span the following interleaving policies gathered from published research:

- **Row**: The LSB bits of the memory address decide the Column index, followed by the Row index, Bank, Rank and the MSB bits decide the Channel (ChRaBaRoCo).
- **Channel**: This address interleaving is devised with the aim of boosting memory level parallelism when a *binary* coded memory address is presented (RoBaRaCoCh).
- **MinOp**: This address interleaving splits the column indices such that exactly 4 consecutive cache lines (assuming a *binary* coded memory address) may map to a single row, thereby striking a balance between row buffer hit rate and memory level parallelism [46].
- **XOR_***: These apply a permutation-based interleaving scheme (for reasons similar to the above) to each of the interleavings presented above [85], [118]. The essence of this is to set the bank index by XORing the bank bits with a selection of higher order bits.

V. Experimental Evaluation

A. Evaluation Methodology

We use pin [65] to generate a dynamic instruction trace that records the program counter of each instruction, as well as any load/store address. Our pintool instruments all 32 bit x86 gcc (-O2) optimized binaries from the SPEC 2006 [20] benchmark suite, with test inputs. These traces drive a ramulator [54] based simulator, complete with an L1, L2, L3 non-inclusive cache hierarchy and DRAM main memory. The baseline configuration is presented in Table II.

Table II: Baseline Configuration

Parameter	Dimensions
Core	Inorder / OoO
OoO Fetch/Retire width/ROB size	4/4/128
L1 size/associativity	32kB/8-way
L2 size/associativity	256kB/8-way
L3 size/associativity	2MB/8-way
Load to use latency L1/L2/L3	4/4+12/4+12+31 cycles
MSHR per cache	16
Caching policy	Non-inclusive/LRU
Core-Memory frequency ratio	3:1
DRAM JEDEC Standard	DDR4 (1channel) / LPDDR4 (2ch)
DRAM address interleaving	Section IV-C
DRAM policy	FRFCFS_prioritizeHit Open page
Memory Address Interpreter	Section IV-B
CLB size	128/1024 entries
CLB policy	Fully associative/LRU
CLB hit / miss latency	1 cycle/binary conversion

We enhance the simulator to support the design space presented in Section IV, and use McPat [59]/CACTI [60] to model energy overheads due to the Memory Address Interpreter. Recall from Section II-C that memory addressing logic such as the MAI is assumed to be error-free and can therefore be modeled with conventional MOSFET-based circuits. For the purposes of the power and area model for the MAI, we assume a 32nm/300K/0.9V/2GHz configuration.

B. Cache

First, we demonstrate the intuition formed in the theoretical analysis of Section III regarding the impact RNS addressing schemes have on locality. Figure 4 shows the cache misses per kilo instructions (MPKI) of RNS schemes, when normalized to binary. *Binary*, captured by **IBIN**, is the normalizing baseline; *rns_concat* is captured by **NRNS**, with the effect of introducing the SELECT instruction captured via **CRNS**, and **SUB** captures behavior of *rns_sub*. The other MAI configurations from Section IV-B do not introduce any new addressing schemes.

As expected from Section III, **NRNS** suffers a significant (18×/13× for L1/L3) loss in spatial locality, whereas **CRNS** helps salvage this. **SUB** was designed to retain the spatial

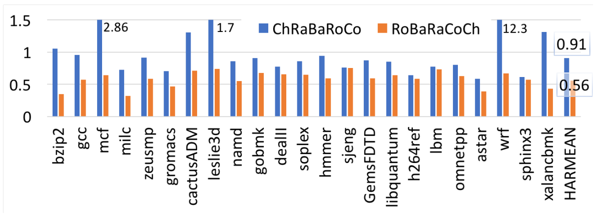
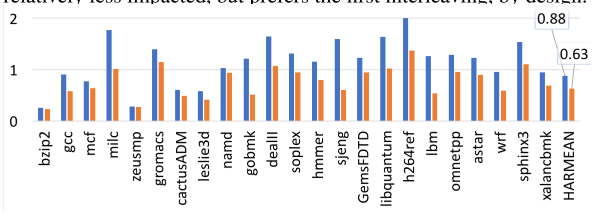
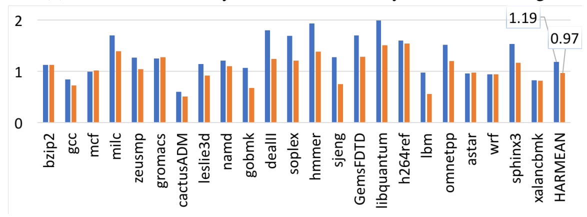


Figure 5: DRAM row buffer hit rate (higher is better) of **SUB**, normalized to **IBIN**, under two example address interleavings. Row locality is completely lost in **CRNS** irrespective of interleaving, and is therefore omitted from this plot. Row locality for **SUB** is relatively less impacted, but prefers the first interleaving, by design.



(**a**) DRAM read latency normalized to binary: *Row* interleaving.



(**b**) DRAM read latency normalized to binary: *Channel* intlv.

Figure 6: DRAM read latency (lower is better) of representative RNS based schemes, normalized to a binary scheme, under two example address interleavings. RNS based schemes naturally extract more memory level parallelism. Together with row locality characteristics (Figure 5), read latency of RNS based schemes are about on-par with or better than binary based schemes.

locality that is present in the lower order bits of **IBIN**, and successfully, is more or less on par with it. The interplay of addressing and temporal locality is rendered responsible for cases where RNS schemes show superior cache performance to binary. Results from cache configuration sweeps w.r.t. size/associativity/replacement policy are as expected. Therefore, we omit them as it adds no new insight to the architecture community.

C. DRAM

There are two aspects to DRAM performance: row buffer hit rate and memory level parallelism exploited.

By design, we expect **SUB** to show somewhat similar row buffer hit rate to **IBIN** for an interleaving policy that respects the locality preserving design principle of **SUB**, such as *row* interleaving. Figure 5 shows its hit rate, normalized to that of binary, for two example interleaving policies. **CRNS**, however, preserves spatial locality at a smaller granularity, thereby exhibiting very low row buffer hit rate. For brevity, we do not present detailed results for **NRNS** because of its incredibly poor cache performance, and omit depiction of **CRNS** from Figure 5 because of its near zero hit rate. The interleaving policy has a significant impact on row buffer locality.

Table III: Most favored DRAM interleaving policy for an MAI configuration. In general, RNS based configurations (such as **NRNS**) benefit from increased memory level parallelism especially when linear address interleaving such as *row* interleaving are used and binary based configurations (such as **IBIN**) benefit from permuted and *XOR* interleavings. Favorable interleavings choice is also affected by row buffer locality and memory pressure differences (due to interplay between MAI configuration, cache performance and processor configuration). If several equally performant choices are available, an arbitrary selection is made.

	MAI Configuration	Inorder	OoO
Ideal	IBIN	XOR_MinOp	XOR_MinOp
Naive	NL1	XOR_Channel	
	NRNS	Row	Row
	NMEM		
Rns_sub	SUB	XOR_MinOp	XOR_MinOp
	SUBM	Row	Row
Compiler	CRNS	XOR_MinOp	XOR_MinOp
	CL2	Row	Row
	CL3	XOR_MinOp	XOR_MinOp
	CMEM		
Compiler / CLB	CCLBMEM_128		
	CCLBMEM_1024		
CLB	CLBL1_128		
	CLBL1_1024	XOR_Channel	
	CLBMEM_128	Row	Row
	CLBMEM_1024		
Rns_sub / CLB	CLBS_128	XOR_MinOp	XOR_MinOp
	CLBS_1024		

On the other hand, because RNS addressing *naturally* permutes an address, a higher degree of memory level parallelism is expected, especially for linear interleaving policies such as *row* interleaving. Figure 6 demonstrates this, as we observe that the average read latency is significantly improved inspite of an inferior row buffer hit rate for RNS based schemes. This is one of the reasons why several interleaving policies (Section IV-C) are deployed for binary based addressing systems, i.e., permuting the address bits reduces bank conflicts and therefore boosts performance (other reasons not related to performance improvement for such permutation are to avoid row-hammer [2] and security issues).

The results presented in this section are with an inorder processor, but similar trends are seen for OoO as well. Also, DRAM scheduling and paging policies have an insignificant impact on performance when compared to the impact due to address interleaving. Therefore we omit their results for brevity.

D. Overall Runtime

For completeness, we simulate the exhaustive set of 18 MAI configurations from Section IV-B across all 6 DRAM address interleavings from Section IV-C, and for presentation purposes summarize the performance for each configuration with its most favored DRAM interleaving policy (Table III) in Figure 7. In deriving the most favored interleaving, no distinction is made between workloads, i.e., the interleaving policy is varied only with MAI configuration and is workload independent.

We normalize their performances against a fixed baseline (**NL1** with *row* interleaving) to be able to compare across configurations and interleavings, and present the resulting speedups. We present only the harmonic mean across the benchmark suite.

[**Ideal**] For example, a conventional non-error-correcting binary core **IBIN** performs 2.84×/2.13× faster on average when compared to the baseline (for inorder/OoO processor respectively, with a DDR4 DRAM), and favors XOR_MinOp as its preferred address interleaving.

[**Naive**] None of the alternate address interleavings make a noticeable improvement to the performance of the baseline

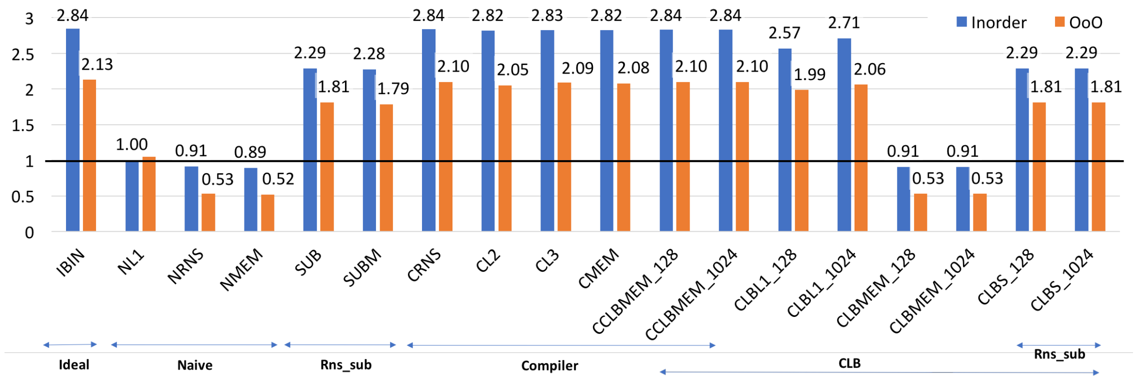
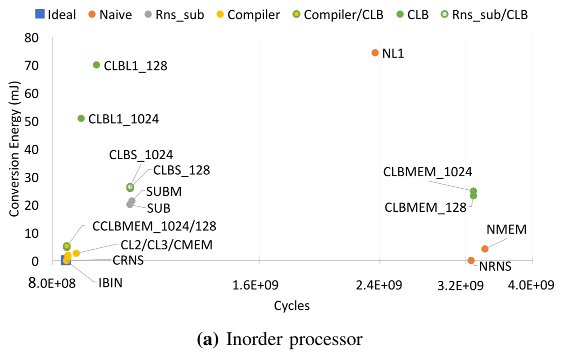


Figure 7: Speedup (higher is better) when compared to a naive approach of converting to binary upon each L1 access (**NL1** with *row* interleaving). Inorder cores are understandably more sensitive to conversion latency than OoO cores. For both inorder/OoO cores, the relative performance of each cluster is **Ideal** > **Compiler** > **CLB** > **Rns_sub** >> **Naive**.



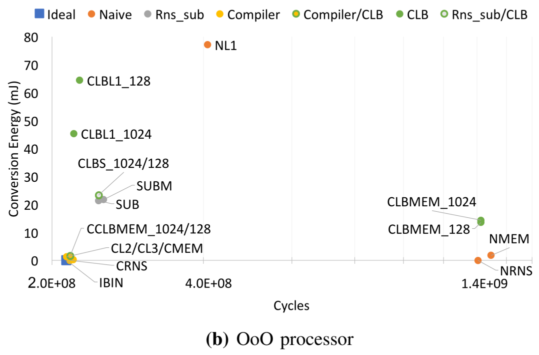


Figure 8: Overhead in conversion energy (mJ) vs runtime; lower is better for both axes (similar to a pareto chart). **SUB** is the most efficient microarchitecture configuration that requires no support from the software stack.

NL1. The 8 cycle conversion latency on every access to the memory hierarchy is left unhidden, allowing it to dominate the performance trends. **NRNS/NMEM** incur significant slowdowns because of their poor cache performance.

[**Rns_sub**] **SUB** shows significantly improved performance over the naive approaches. **SUBM** follows closely behind, due to slight decrease in exploited memory level parallelism, coupled with its conversion latency at the memory controller.

[**Compiler**] The approaches that leverage the SELECT instruction effectively approach **IBIN** by combining the best of binary (cache performance) and RNS (no conversion overhead).

[**CLB**] Placing a 128 entry CLB before L1 allows for performance superior to **SUB**, and a 1024 entry CLB seems sufficient to approach the performance of **IBIN**. However, placing a CLB at the memory controller is rendered useless unless either the SELECT instruction is used or *rns_sub* is used to prevent an explosion of cache misses.

From an Amdahl's law perspective, the amount of overhead and variation in memory access patterns introduced by various MAI configurations has more weight than just the DRAM interleaving policy, which is to be expected. If we were to choose an unfavorable DRAM interleaving instead, the performance on average for each of the non-naive configurations vary by less than 1%, although we omit detailed results for brevity.

LPDDR4 is a likely candidate to be used in conjunction with low power microarchitectures, but comes at a cost of increased row activation latency. Nevertheless, we find that the insights presented in this paper hold even when an LPDDR4 DRAM is used.

E. Energy Overhead

Figure 8 presents the performance of each of the MAI configurations, when put in perspective of how much *additional* energy they consume. For each MAI configuration, their most favored DDR4-DRAM interleaving is chosen (Table III) and the mean cycle count and conversion energy overhead (mJ) across the benchmark suite are presented.

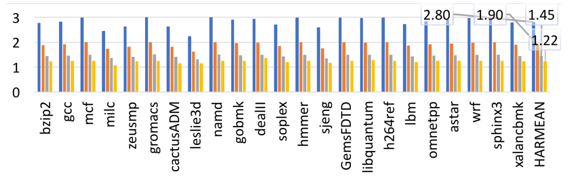
[**Ideal**] For example, at the bottom left is **IBIN**, taking the least number of cycles and without any conversion overhead.

[**Naive**] **NL1**, as has been mentioned throughout this paper, suffers from poor performance and high energy consumption. **NMEM/NRNS** have little or no increase in energy consumption due to conversion alone, but their performance suffers greatly.

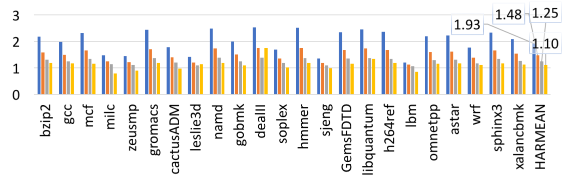
[**Rns_sub**] **SUB/SUBM** are the most *efficient* microarchitectures that do not require support from the software stack for them to work.

[**Compiler**] While these approach **IBIN**, they are subject to software compatibility and integration issues as discussed in Section IV-A.

[**CLB**] While placing a CLB at the L1 is more performant than **SUB**, it comes at a higher conversion energy overhead. Placing a CLB at the memory controller must be accompanied by either an *rns_sub* addressed cache or a cache line addressed (SELECT instruction) cache.



(a) Inorder processor



(b) OoO processor

Figure 9: Slowdown of hypothetically faster binary convertors (4 cycles and 2 cycles as opposed to the best known candidate: 8 cycles) when compared to **IBIN**. **NL1** thats used everywhere in this document is annotated as **NL1_8** for clarity. Our most efficient microarchitecture technique that uses **SUB** is still faster than these hypothetically faster convertors. It is also more energy efficient than these, however, a quantitative comparison is not possible in the absence of concrete algorithms for faster binary conversion.

Deriving a CLB configuration is similar to that of a TLB- or cache-like structure. For example, while increasing the CLB size increases access power, it may reduce the number of CLB misses, which also translates into energy savings on CLB miss repairs (ex: **CLBL1_128** vs **CLBL1_1024**). For brevity, we limit a CLB configuration sweep to just these two for demonstrative purposes; the reader should be able to easily infer points of tradeoff for other CLB configurations.

MAI Area. Table IV presents the area requirements of few key configurations. We conclude that the energy trends discussed above apply to area as well.

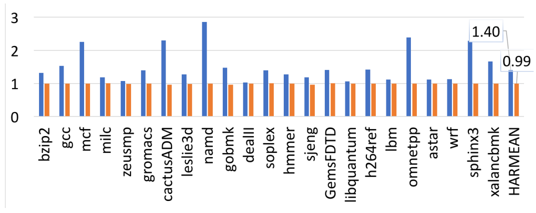
Table IV: Area requirement in mm^2.

NRNS	SUB	CLB_128	CLB_1024	NL1
0	0.075	0.091	0.104	0.160

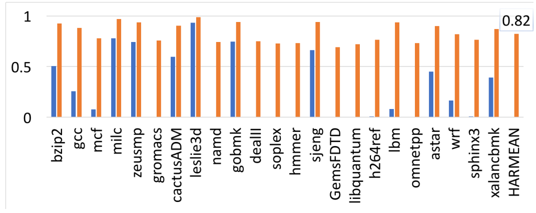
F. Sensitivity to Conversion Latency

Throughout this paper, we have assumed an 8 cycle algorithm for converting an RNS tuple to a binary number (**NL1**), as it is the state of the art given that the bases must be amenable to RRNS error correction (Section II-B). Nevertheless, we also evaluate the relative performance of hypothetical conversion algorithms that take half as many or even a quarter of the cycles. For clarity, we differentiate these via subscripts (**NL1_8**, **NL1_4**, **NL1_2**). Figure 9 puts their performance in perspective of two representative MAI configurations: **IBIN** and **SUB**. Recall that **SUB** presents the most efficient microarchitecture that does not impose restrictions on software. Therefore, for this analysis, we choose **IBIN** as the baseline and also present **SUB** and **NL1_8** to put the performance of these hypothetical convertors into perspective. Relative performance of the remaining MAI configurations can be easily inferred from the previous result sections.

Not only are these hypothetical faster convertors less performant than the schemes presented in this paper, but would also be less energy efficient. While a quantitative comparison is not possible in the absence of concrete faster algorithms, we expect their energy overhead to be rather close to **NL1_8** (and certainly much more than **SUB**) from a functional standpoint of the process of converting to binary.



(a) TLB hit rate normalized to binary (higher is better)



(b) Fraction of pages that do not cause a hard disk access, normalized to binary (higher is better)

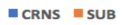


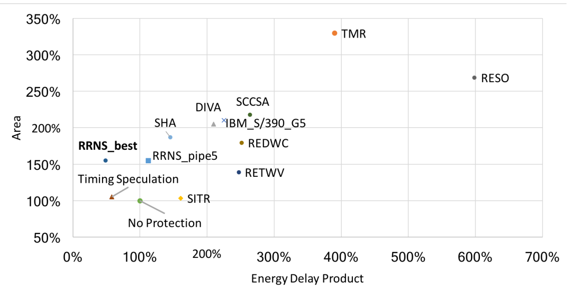
Figure 10: Virtual memory performance of **CRNS** and **SUB** when compared to **IBIN**. **CRNS** has superior TLB hit rate owing to its higher memory access granularity, however, an explosion is seen in the number of physical pages it touches, owing to its limited spatial locality granularity. As expected, both TLB and page pressure of **SUB** approach that of **IBIN**.

G. Virtual Memory

Certain RRNS based architectures may find it necessary to integrate with a virtual memory (VM) subsystem. There are two aspects to VM performance: TLB hit rate and page pressure. Under an idempotent virtual-to-physical transformation, we use the reuse distance [7], [12] mechanism to estimate the TLB hit rate in a 512kB structure and to estimate page pressure using an 8GB DRAM, assuming a page size of 4kB (as is common on Linux systems). We quantify page pressure by measuring the number of pages that have already been brought into main memory as well as the number of pages that need to access secondary memory (such as a non-volatile disk), as they may either have not yet been allocated a physical page frame or had been evicted to make way for newer pages.

Figure 10 highlights these for representative RNS schemes, when compared to binary. Given its 64 byte access granularity, **CRNS** exhibits high TLB performance thanks to sequential locality in programs. However, this granularity is rather small at the page level, causing an explosion in the number of pages it accesses. As described in Section IV-A, **CRNS** would have to be extended into the runtime system (or a different page granularity should be used) in order to attain low page pressure.

SUB, on the other hand, performs very similar to **IBIN** on both counts. The reason for its relatively higher page pressure when compared to binary is that 4kB pages have an offset of 12 bits, whereas the residues in this paper are (or specifically the fourth residue) 8 bits wide, thereby leading to a gap in locality between *rns_sub* and *binary*. Choosing a wider m_4 would result in an interesting tradeoff as it may change the power-performance-reliability characteristics of the RRNS core, but from a memory systems point of view, it would enhance the spatial locality properties of *rns_sub* which directly translates to improved cache performance, DRAM row buffer hit rate and virtual memory performance.



RESO [78], REDWC [45], RETWV [41], SCCSA [108], SHA [79], DIVA [1], SITR [70], Timing Speculation [26], [37]

Figure 11: First order comparison of area overhead and energy-delay product (EDP) of various mechanisms for computational error correction, depicting the superiority of RRNS. Computational error correction techniques use a combination of spatial and temporal redundancy techniques. While temporal redundancy allows for a low area overhead, they suffer from a significant performance penalty. Timing speculation techniques seem more efficient than RRNS, however, their error model assumes all bit errors manifest as circuit timing errors, which is not sufficient to work with ultra low energy logic devices.

VI. Related Work

Computational error correction. Figure 11, Section I-A summarize various techniques in comparison with RRNS. We refer the interested reader to Srikanth et al. [91] for a more detailed survey on some of these non-residue techniques; RRNS is generally considered superior in terms of capability and efficiency for computational error resilience. State of the art adoption and research in the industry also claim the superiority of residue based resilience [16], [61].

Approaches that employ timing speculation [26], [37] may seem superior to RRNS at first glance. However, the error model that can be supported by an RRNS error correcting architecture is orthogonal to theirs, if not broader. For example, razor [26] uses conventional transistors, therefore lowering V_{dd} lowers MOSFET switching speed significantly, resulting in a large frequency drop, which could cause setup time violations that they handle via a delayed latch mechanism. They assume that any error manifests itself as a timing error. Similarly, decor [37] uses a delayed commit approach (with rollback support) to handle violations in timing margins. However, with emerging devices (Section I), V_{dd} can be lowered to few tens of millivolts with a relatively lower frequency loss, meaning that operating at the resultant thermal noise floor leads to *stochastic, intermittent* bit flips, which cannot always be captured as circuit timing errors. Unlike such approaches, RRNS error correcting architectures can not only tolerate such errors in the data path, but also in the control path between memory accesses.

In terms of being able to tolerate control path errors, approaches such as DIVA [1] that replicate parts of the pipeline are capable. Their design provides recovery by having a simple core recalculate results of an out-of-order core. In this approach, the simple core is assumed to be error-free. This is similar to a ”double-modular-redundancy” approach with a rad-hard node, implying a relatively high overhead. Furthermore, if the rad-hard simple core is instead prone to error, checkpoint and re-execute methods would need to be employed, similar to the IBM POWER6/7/8 and z10/z196 processors [8], [16], [44], [61] and various Intel Corporation [90] and Sun Microsystems based mainframes [43]. On the other hand, RRNS is able to tolerate errors in its redundant as well as non-redundant computations.

Finally, a vast majority of these related work are at the circuit level and can be augmented into RRNS-based architectures for potentially enhanced reliability characteristics.

Prime number based indexing. Kharbutli et al. [53] propose using prime numbers for cache indexing to reduce conflict misses. However, such indexing introduces fragmentation that can only be amortized by higher cache capacities. They therefore recommend using their technique only for larger caches (such as L2/L3). Furthermore, their technique isn't applicable to DRAM addressing.

VII. Conclusion

New logic devices enable reduction of the supply voltage to few tens of millivolts, yet maintaining a switching speed superior to MOSFETs under low power operation. However, they are subject to intermittent, stochastic bit errors in logic due to proximity of operation to the kT noise floor. Computational error correction using the RRNS representation is a promising approach to using these devices. However, prior RRNS based architecture studies assume a simple, unrealistic memory hierarchy without considering issues with RNS-based memory addressing. This research found that naive approaches such as **NL1** and **NRNS** incur an overhead of $2\times$-$4\times$ on average. We propose new conversion strategies and architecture extensions to significantly improve on naive memory system performance. In our classification of the design space, the relative performance is **Ideal** $>$ **Compiler** $>$ **CLB** $>$ **Rns_sub** $>>$ **Naive**, with **Rns_sub** rendering the most energy efficient architecture along with the added advantage of requiring no support from the software stack.

The careful analysis of RNS-based memory access pattern behavior and the detailed cost benefit analysis of the resulting design space presented in this paper enables and extends RRNS-error-correcting core microarchitectures built from ultra-low energy logic devices. Using such devices has the potential to punch through the power wall to restart single core performance scaling via architectural advances abandoned at the end of Dennard scaling a decade ago.

VIII. Appendix: SELECT Instruction

Consider an implementation with a 64 byte cache line size, a *double* of size 8 bytes such that the following *struct* has a size of 16 bytes, and the following snippet of code, where $M < N$ are arbitrary natural numbers that are not necessarily compile-time constants.

```
typedef struct { double x; double y; } Foo;
Foo foos[N];
double sum = 0;
for (i = 0; i < M; ++i) {
  sum += foos[i].x;
  sum += foos[i].y;
}
```

A pseudo-assembly code of the loop body in a *binary* computer would be as follows:

```
LD X, (foos + i*16)     // Read foos[i].x
LD Y, (foos + i*16 + 1*8)  // Read foos[i].y
ADD S, S, X
ADD S, S, Y
```

When SELECT instruction is used, the code transforms to:

```
uint8_t nOffset = sizeof(CACHELINE)/sizeof(Foo); //
     64/16=4
// ++i is in RRNS ; ++n is in binary
for (rns_t i = 0; i < M / nOffset; ++i) {
  LD A64, RNS(foos + i*sizeof(CACHELINE))
  for (int_t n = 0; n < nOffset; ++n) {
    SELECT X, A64, sizeof(Foo)*n
    SELECT Y, A64, sizeof(Foo)*n + 1*8
    ADD S, S, X
    ADD S, S, Y
  }
}
```

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-NA0003525.

References

[1] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*. IEEE, 1999, pp. 196–207.

[2] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield, "Row hammer refresh command," Aug. 25 2015, uS Patent 9,117,544.

[3] J.-C. Bajard and L. Imbert, "A full rns implementation of rsa," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 769–774, 2004.

[4] J.-C. Bajard, L.-S. Didier, and J.-M. Muller, "A new euclidean division algorithm for residue number systems," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 19, no. 2, pp. 167–178, 1998.

[5] J.-C. Bajard, J. Eynard, and N. Merkiche, "Multi-fault attack detection for rns cryptographic architecture," in *Computer Arithmetic (ARITH), 2016 IEEE 23nd Symposium on*. IEEE, 2016, pp. 16–23.

[6] F. Barsi and P. Maestrini, "Error detection and correction by product codes in residue number systems," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 915–924, 1974.

[7] B. T. Bennett and V. J. Kruskal, "Lru stack processing," *IBM Journal of Research and Development*, vol. 19, no. 4, pp. 353–357, 1975.

[8] M. J. Boersma and J. Haess, "Residue-based error detection for a processor execution unit that supports vector operations," Mar. 17 2015, uS Patent 8,984,039.

[9] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations," *IRE Transactions on Electronic Computers*, no. 3, pp. 333–337, 1960.

[10] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[11] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3. ACM, 1991, pp. 276–286.

[12] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 2003, pp. 150–159.

[13] B. H. Calhoun, A. Wang, and A. Chandrakasan, "Modeling and sizing for minimum energy operation in subthreshold circuits," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 9, pp. 1778–1786, 2005.

[14] B. Cao, C.-H. Chang, and T. Srikanthan, "A residue-to-binary converter for a new five-moduli set," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 5, pp. 1041–1049, 2007.

[15] G. Cardarilli, M. Re, and R. Lojacono, "Rns-to-binary conversion for efficient vlsi implementation," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 45, no. 6, pp. 667–669, 1998.

[16] S. Carlough, A. Collura, S. Mueller, and M. Kroener, "The ibm zenterprise-196 decimal floating-point accelerator," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*. IEEE, 2011, pp. 139–146.

[17] C.-H. Chang, A. S. Molahosseini, A. A. E. Zarandi, and T. F. Tay, "Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications," *IEEE circuits and systems magazine*, vol. 15, no. 4, pp. 26–44, 2015.

[18] J.-S. Chiang and M. Lu, "Floating-point numbers in residue number systems," *Computers &amp; Mathematics with Applications*, vol. 22, no. 10, pp. 127–140, 1991.

[19] R. Chokshi, K. S. Berezowski, A. Shrivastava, and S. J. Piestrak, "Exploiting residue number system for power-efficient digital signal processing in embedded processors," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2009, pp. 19–28.

[20] S. CPU2006, "Standard performance evaluation corporation," 2006.

[21] B. Deng, S. Srikanth, E. R. Hein, P. G. Rabbat, T. M. Conte, E. DeBenedictis, and J. Cook, "Computationally-redundant energy-efficient processing for y'all (creepy)," in *Rebooting Computing (ICRC), IEEE International Conference on*. IEEE, 2016, pp. 1–8.

[22] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[23] E. D. Di Claudio, G. Orlandi, and F. Piazza, "A systolic redundant residue arithmetic error correction circuit," *IEEE Transactions on Computers*, vol. 42, no. 4, pp. 427–432, 1993.

[24] E. D. Di Claudio, F. Piazza, and G. Orlandi, "Fast combinatorial rns processors for dsp applications," *IEEE transactions on computers*, vol. 44, no. 5, pp. 624–633, 1995.

[25] S. Dolev, S. Frenkel, D. E. Tamir, and V. Sinelnikov, "Preserving hamming distance in arithmetic and logical operations," *Journal of Electronic Testing*, vol. 29, no. 6, pp. 903–907, 2013.

[26] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 7–18.

[27] M. Etzel and W. Jenkins, "Redundant residue number systems for error detection and correction in digital filters," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 5, pp. 538–545, 1980.

[28] C. Fetzer, U. Schiffel, and M. Süßkraut, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2009, pp. 283–296.

[29] P. Forin, "Vital coded microprocessor principles and application for various transit systems," *IFAC Control, Computers, Communications*, pp. 79–84, 1989.

[30] D. Gamberger, "New approach to integer division in residue number systems," in *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*. IEEE, 1991, pp. 84–91.

[31] H. L. Garner, "The residue number system," *IRE Transactions on Electronic Computers*, no. 2, pp. 140–147, 1959.

[32] S. Ghosh, P. Ndai, and K. Roy, "A novel low overhead fault tolerant kogge-stone adder using adaptive clocking," in *Design, Automation and Test in Europe, 2008. DATE'08*. IEEE, 2008, pp. 366–371.

[33] V. T. Goh and M. U. Siddiqi, "Multiple error detection and correction based on redundant residue number systems," *IEEE Transactions on Communications*, vol. 56, no. 3, 2008.

[34] O. Goldreich, D. Ron, and M. Sudan, "Chinese remaindering with errors," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*. ACM, 1999, pp. 225–234.

[35] J. González and A. González, "Limits of instruction level parallelism with data speculation," in *Proc. of the VECPAR Conf.* Citeseer, 1998, pp. 585–598.

[36] K. C. Gower, B. Hazelzet, M. W. Kellogg, and D. J. Perlman, "High reliability memory module with a fault tolerant address and command bus," Jun. 19 2007, uS Patent 7,234,099.

[37] M. S. Gupta, K. K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, "Decor: A delayed commit and rollback mechanism for handling inductive noise in processors," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 2008, pp. 381–392.

[38] N. Z. Haron and S. Hamdioui, "Redundant residue number system code for fault-tolerant hybrid memories," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 7, no. 1, p. 4, 2011.

[39] A. A. Hiasat and H. Abdel-Aty-Zohdy, "Semi-custom vlsi design and implementation of a new efficient rns division algorithm," *The Computer Journal*, vol. 42, no. 3, pp. 232–240, 1999.

[40] M. A. Hitz and E. Kaltofen, "Integer division in residue number systems," *IEEE transactions on computers*, vol. 44, no. 8, pp. 983–989, 1995.

[41] Y.-M. Hsu and E. Swartzlander, "Time redundant error correcting adders and multipliers," in *Defect and Fault Tolerance in VLSI Systems, 1992. Proceedings., 1992 IEEE International Workshop on*. IEEE, 1992, pp. 247–256.

[42] C. Y. Hung and B. Parhami, "Fast rns division algorithms for fixed divisors with application to rsa encryption," *Information Processing Letters*, vol. 51, no. 4, pp. 163–169, 1994.

[43] S. Iacobovici, "End-to-end residue based protection of an execution pipeline," Jun. 30 2009, uS Patent 7,555,692.

[44] IBM, "Ibm power system e880 server, an ibm power8 technology-based system, addresses the requirements of an industry-leading enterprise class system," 2014.

[45] B. W. Johnson, J. H. Aylor, and H. H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit vlsi adder," *IEEE journal of solid-state circuits*, vol. 23, no. 1, pp. 208–215, 1988.

[46] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A dram page-mode scheduling policy for the many-core era," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 24–35.

[47] R. S. Katti, "A new residue arithmetic error correction scheme," *IEEE transactions on computers*, vol. 45, no. 1, pp. 13–19, 1996.

[48] O. Keren, I. Levin, V. Ostrovsky, and B. Abramov, "Arbitrary error detection in combinational circuits by using partitioning," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*. IEEE, 2008, pp. 361–369.

[49] R. Keyes, "Miniaturization of electronics and its limits," *IBM Journal of Research and Development*, vol. 32, no. 1, pp. 84–88, 1988.

[50] A. I. Khan, C. W. Yeung, C. Hu, and S. Salahuddin, "Ferroelectric negative capacitance mosfet: Capacitance tuning &amp; antiferroelectric operation," in *Electron Devices Meeting (IEDM), 2011 IEEE International*. IEEE, 2011, pp. 11–3.

[51] A. I. Khan, K. Chatterjee, J. P. Duarte, Z. Lu, A. Sachid, S. Khandelwal, R. Ramesh, C. Hu, and S. Salahuddin, "Negative capacitance in short-channel finfets externally connected to an epitaxial ferroelectric capacitor," *IEEE Electron Device Letters*, vol. 37, no. 1, pp. 111–114, 2016.

[52] A. I. Khan and S. Salahuddin, "4 extending cmos with negative capacitance," *CMOS and Beyond: Logic Switches for Terascale Integrated Circuits*, pp. 56–76, 2015.

[53] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *Software, IEE Proceedings-*. IEEE, 2004, pp. 288–299.

[54] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[55] E. Krekhov, A.-r. A. Pavlov, A. Pavlov, P. Pavlov, D. Smirnov, A. Tsar'kov, P. Chistopol'skii, A. Shandrikov, B. Sharikov, and D. Yakimov, "A method of monitoring execution of arithmetic operations on computers in computerized monitoring and measuring systems," *Measurement Techniques*, vol. 51, no. 3, pp. 237–241, 2008.

[56] H. Krishna, B. Krishna, K.-Y. Lin, and J.-D. Sun, *Computational Number Theory and Digital Signal Processing: Fast Algorithms and Error Control Techniques*. CRC Press, 1994, vol. 6.

[57] H. Krishna, K.-Y. Lin, and J.-D. Sun, "A coding theory approach to error control in redundant residue number systems. i. theory and single error correction," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 1, pp. 8–17, 1992.

[58] H.-H. Lee, Y. Wu, and G. Tyson, "Quantifying instruction-level parallelism limits on an epic architecture," in *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*. IEEE, 2000, pp. 21–27.

[59] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 469–480.

[60] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. IEEE, 2011, pp. 694–701.

[61] D. Lipetz and E. Schwarz, "Self checking in current floating-point units," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*. IEEE, 2011, pp. 73–76.

[62] C.-K. Liu, "Error-correcting-codes in computer arithmetic," DTIC Document, Tech. Rep., 1972.

[63] H.-Y. Lo and T.-W. Lin, "Parallel algorithms for residue scaling and error correction in residue arithmetic," *Wireless Engineering and Technology*, vol. 4, no. 04, p. 198, 2013.

[64] M. Lu and J.-S. Chiang, "A novel division algorithm for the residue number system," *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 1026–1032, 1992.

[65] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[66] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. Elsevier, 1977.

[67] D. Marienfeld, E. S. Sogomonyan, V. Ocheretnij, and M. Gossel, "New self-checking output-duplicated booth multiplier with high fault coverage for soft errors," in *Test Symposium, 2005. Proceedings. 14th Asian*. IEEE, 2005, pp. 76–81.

[68] J. Mathew, S. Banerjee, P. Mahesh, D. Pradhan, A. Jabir, and S. Mohanty, "Multiple bit error detection and correction in gf arithmetic circuits," in *Electronic System Design (ISED), 2010 International Symposium on*. IEEE, 2010, pp. 101–106.

[69] A. McMenamin, "The end of dennard scaling," 2013.

[70] E. Mizan, T. Amimeur, and M. F. Jacome, "Self-imposed temporal redundancy: An efficient technique to enhance the reliability of pipelined functional units," in *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. IEEE, 2007, pp. 45–53.

[71] P. V. A. Mohan, "Rns-to-binary converter for a new three-moduli set," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 9, pp. 775–779, 2007.

[72] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 89–104, 2002.

[73] M. Nicolaidis, "Carry checking/parity prediction adders and alus," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 1, pp. 121–128, 2003.

[74] M. Nicolaidis and H. Bederr, "Efficient implementations of self-checking multiply and divide arrays," in *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings*. IEEE, 1994, pp. 574–579.

[75] M. Nicolaidis and R. Duarte, "Design of fault-secure parity-prediction booth multipliers," in *Design, Automation and Test in Europe, 1998., Proceedings*. IEEE, 1998, pp. 7–14.

[76] D. E. Nikonov and I. A. Young, "Overview of beyond-cmos devices and a uniform methodology for their benchmarking," *Proceedings of the IEEE*, vol. 101, no. 12, pp. 2498–2533, 2013.

[77] G. A. Orton, L. E. Peppard, and S. E. Tavares, "New fault tolerant techniques for residue number systems," *IEEE transactions on computers*, vol. 41, no. 11, pp. 1453–1464, 1992.
[78] J. H. Patel and L. Y. Fung, "Concurrent error detection in alu's by recomputing with shifted operands," *IEEE Trans. Computers*, vol. 31, no. 7, pp. 589–595, 1982.
[79] S. Peng and R. Manohar, "Fault tolerant asynchronous adder through dynamic self-reconfiguration," in *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 2005, pp. 171–178.
[80] V. Ramachandran, "Single residue error correction in residue number systems," *IEEE transactions on computers*, vol. 32, no. 5, pp. 504–507, 1983.
[81] J. Ramirez, A. Garcia, S. Lopez-Buedo, and A. Lloris, "Rns-enabled digital signal processor design," *Electronics Letters*, vol. 38, no. 6, pp. 266–268, 2002.
[82] T. R. Rao, "Biresidue error-correcting codes for computer arithmetic," *IEEE Transactions on computers*, vol. 100, no. 5, pp. 398–402, 1970.
[83] W. Rao and A. Orailoglu, "Towards fault tolerant parallel prefix adders in nanoelectronic systems," in *Design, Automation and Test in Europe, 2008. DATE'08*. IEEE, 2008, pp. 360–365.
[84] W. Rao, A. Orailoglu, and R. Karri, "Fault identification in reconfigurable carry lookahead adders targeting nanoelectronic fabrics," in *Test Symposium, 2006. ETS'06. Eleventh IEEE European*. IEEE, 2006, pp. 63–68.
[85] B. R. Rau, "Pseudo-randomly interleaved memory," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3. ACM, 1991, pp. 74–83.
[86] M. Sachdev, "Fault-tolerant memory address decoder," Nov. 3 1998, uS Patent 5,831,986.
[87] S. Salahuddin and S. Datta, "Can the subthreshold swing in a classical fet be lowered below 60 mv/decade?" in *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*. IEEE, 2008, pp. 1–4.
[88] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "Anb- and anbdmem-encoding: detecting hardware errors in software," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2010, pp. 169–182.
[89] A. Sengupta and B. Natarajan, "Performance of systematic rrns based space-time block codes with probability-aware adaptive demapping," *IEEE Transactions on Wireless Communications*, vol. 12, no. 5, pp. 2458–2469, 2013.
[90] Z. Sperber, O. Levy, M. Mishaeli, and R. Gabor, "Recoverable parity and residue error," Dec. 9 2014, uS Patent 8,909,988.
[91] S. Srikanth, B. Deng, and T. M. Conte, "A brief survey of non-residue based computational error correction," *arXiv preprint arXiv:1611.03099*, 2016.
[92] C.-C. Su and H.-Y. Lo, "An algorithm for scaling and single residue error correction in residue number systems," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1053–1064, 1990.
[93] J.-D. Sun and H. Krishna, "A coding theory approach to error control in redundant residue number systems. ii. multiple error detection and correction," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 1, pp. 18–34, 1992.
[94] J.-D. Sun, H. Krishna, and K. Lin, "A superfast algorithm for single-error correction in rrns and hardware implementation," in *Circuits and Systems, 1992. ISCAS'92. Proceedings., 1992 IEEE International Symposium on*, vol. 2. IEEE, 1992, pp. 795–798.
[95] Y. Sun, M. Zhang, S. Li, and Y. Zhao, "Cost effective soft error mitigation for parallel adders by exploiting inherent redundancy," in *IC Design and Technology (ICICDT), 2010 IEEE International Conference on*. IEEE, 2010, pp. 224–227.
[96] R. M. Swanson and J. D. Meindl, "Ion-implanted complementary mos transistors in low-voltage circuits," *IEEE Journal of Solid-State Circuits*, vol. 7, no. 2, pp. 146–153, 1972.
[97] A. Sweidan and A. A. Hiasat, "On the theory of error control based on moduli with common factors," *Reliable computing*, vol. 7, no. 3, pp. 209–218, 2001.
[98] N. S. Szabo and R. I. Tanaka, *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.
[99] S. Talahmeh and P. Siy, "Arithmetic division in rns using galois field gf (p)," *Computers &amp; Mathematics with Applications*, vol. 39, no. 5-6, pp. 227–238, 2000.
[100] M. Talluri and M. D. Hill, *Surpassing the TLB performance of superpages with less operating system support*. ACM, 1994, vol. 29, no. 11.
[101] Y. Tang, E. Boutillon, C. Jégo, and M. Jézéquel, "A new single-error correction scheme based on self-diagnosis residue number arithmetic," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*. IEEE, 2010, pp. 27–33.
[102] T. F. Tay and C.-H. Chang, "A new algorithm for single residue digit error correction in redundant residue number system," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 1748–1751.
[103] ——, "A non-iterative multiple residue digit error detection and correction algorithm in rrns," *IEEE transactions on computers*, vol. 65, no. 2, pp. 396–408, 2016.
[104] T. N. Theis, "(keynote) in quest of a fast, low-voltage digital switch," *ECS Transactions, 45(6), 3-11*, 2012.
[105] T. N. Theis and P. M. Solomon, "In quest of the "next switch": prospects for greatly reduced power dissipation in a successor to the silicon field-effect transistor," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2005–2014, 2010.
[106] M. Valinataj and S. Safari, "Fault tolerant arithmetic operations with multiple error detection and correction," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*. IEEE, 2007, pp. 188–196.
[107] D. P. Vasudevan and P. K. Lala, "A technique for modular design of self-checking carry-select adder," in *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on*. IEEE, 2005, pp. 325–333.
[108] D. P. Vasudevan, P. K. Lala, and J. P. Parkerson, "Self-checking carry-select adder design based on two-rail encoding," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 12, pp. 2696–2705, 2007.
[109] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.
[110] J. Von Neumann, A. W. Burks *et al.*, "Theory of self-reproducing automata," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 3–14, 1966.
[111] U. Wappler and C. Fetzer, "Hardware failure virtualization via software encoded processing," in *Industrial Informatics, 2007 5th IEEE International Conference on*, vol. 2. IEEE, 2007, pp. 977–982.
[112] R. W. Watson and C. W. Hastings, "Self-checked computation using residue arithmetic," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1920–1931, 1966.
[113] H. Xiao, H. K. Garg, J. Hu, and G. Xiao, "New error control algorithms for residue number system codes," *ETRI Journal*, vol. 38, no. 2, pp. 326–336, 2016.
[114] L. Xiao and X.-G. Xia, "Error correction in polynomial remainder codes with non-pairwise coprime moduli and robust chinese remainder theorem for polynomials," *IEEE Transactions on Communications*, vol. 63, no. 3, pp. 605–616, 2015.
[115] S.-S. Yau and Y.-C. Liu, "Error correction in redundant residue number systems," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 5–11, 1973.
[116] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon, "Rsa speedup with chinese remainder theorem immune against hardware fault cryptanalysis," *IEEE Transactions on computers*, vol. 52, no. 4, pp. 461–472, 2003.
[117] P. Yin and L. Li, "A new algorithm for single error correction in rrns," in *Communications, Circuits and Systems (ICCCAS), 2013 International Conference on*, vol. 2. IEEE, 2013, pp. 178–181.
[118] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM, 2000, pp. 32–41.

Appendix A

Early and Unpublished RRNS and Superstrider Work

Below is a collection of both published and unpublished work that was done on the RRNS-based architecture and the early PIMS architecture. *Optimal Adiabatic Scaling and Processor-in-Memory-and-Store Architecture (OAS+PIMS)* was published in the 2015 IEEE/ACM International Symposium on Nanoscale Architectures. The papers, titled *Computationally-Redundant Energy-Efficient Processing for Y?all (CREEPY)* and *Energy Efficiency Limits of Logic and Memory*, were published in the IEEE International Conference on Rebooting Computing, 2016. The rest of the work included in this section are short reports that were not published, but were generated as project documentation. Note that the paper titled *Representing real numbers in a Redundant Residual Number System (RRNS) based computer* is a draft that was submitted, but never accepted for publication.

Optimal Adiabatic Scaling and the Processor-In-Memory-and-Storage Architecture (OAS+PIMS)

Erik P. DeBenedictis, Jeanine E. Cook, Mark F. Hoemmen, Tzevetan S. Metodi
Center for Computing Research, Sandia National Laboratories
P. O. Box 5800 M. S. 1319
Albuquerque, NM 87185-1319 USA

Abstract- **We discuss a new approach to computing that retains the possibility of exponential growth while making substantial use of the existing technology. The exponential improvement path of Moore's Law has been the driver behind the computing approach of Turing, von Neumann, and FORTRAN-like languages. Performance growth is slowing at the system level, even though further exponential growth should be possible.**

We propose two technology shifts as a remedy, the first being the formulation of a scaling rule for scaling into the third dimension. This involves use of circuit-level energy efficiency increases using adiabatic circuits to avoid overheating. However, this scaling rule is incompatible with the von Neumann architecture.

The second technology shift is a computer architecture and programming change to an extremely aggressive form of Processor-In-Memory (PIM) architecture, which we call Processor-In-Memory-and-Storage (PIMS). Theoretical analysis shows that the PIMS architecture is compatible with the 3D scaling rule, suggesting both immediate benefit and a long-term improvement path.

I. Introduction

The objective of the paper is to devise a new approach to computing that circumvents what is colloquially called "the end of Moore's Law." We would like to do so in a way that leverages the technology stack currently in use as much as possible, but we do not know how to do this by changing just one level of the stack. However, it appears that changing several levels in concert yields a suitable result.

We propose two new technologies that are each insufficient in isolation, but together seem able to create both an immediate performance improvement for certain classes of applications and an improvement path that continues over time. We summarize the way these technologies combine here; subsequent sections provide a more detailed justification.

There are two problems leading to the perception that Moore's Law is ending: (1) heat per unit area increases under potential Moore's Law scenarios, and (2) continued shrinkage of semiconductor line width is reaching a limit due to both wavelength needed in the lithographic equipment and atomic effects in the devices.

Our first proposed technology, called Optimal Adiabatic Scaling (OAS) for 3D chips, resolves the two problems – but with a caveat. OAS combines power-efficient adiabatic circuits with 3D scaling to achieve constant power per unit area even under scaling. However, OAS will produce 3D chips that require the amount of application-level parallelism to grow as the technology scales. When OAS chips scale, the computational throughput grows and the clock rate drops with the square root of the growth in the number of devices. The von Neumann computer architecture cannot make use of this scaling, leading to the second technology.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The second proposed technology, called Processor-In-Memory-and-Storage (PIMS), is in the familiar Processor-In-Memory class but uses non-volatile devices and is targeted at problems that begin and end with data in storage. Initial versions of a PIMS system may make use of some of emerging advanced memories, such as Resistive Random Access Memory (RRAM) and Phase Change Memory (PCM), but eventually new specialized requirements will likely emerge and are discussed in Section VII.

PIMS is very nearly a general-purpose architecture that can be implemented in a standard computing system. If PIMS is implemented with OAS, it will scale throughput and storage in the same ratio as contemporary *systems* (i.e., it scales as a processor plus disk drive combination, not just the processor).

OAS and PIMS together will not be a complete solution to extending Moore's Law. Some problems simply lack parallelism and require fast sequential execution. However, PIMS can exist on a continuum that trades off thread speed and energy efficiency all the way down to the gate level.

II. Optimal Adiabatic Scaling

We will start with an abstract argument that will create a 3D analogy to Moore's Law, deferring specifics to a later section. Assume that 2D scaling has run its course and the "final" transistors and other devices are being manufactured on a large scale. While the devices will be unchanging, we seek a scaling rule that applies as manufacturing evolves to create the devices at lower cost over time by exploiting the third dimension [1].

There is an "adiabatic" circuit class [2, 3, 4] where energy per gate operation is proportional to clock rate. A specific circuit will be given in Section III, but for now we consider whether using adiabatic circuits and tying the clock rate to scaling could control energy in the proper way. Moore's paper defining Moore's Law [5] stated that "shrinking dimensions on an integrated structure makes it possible to operate the structure at higher speed for the same power per unit area." We address whether we can use progressively lower speed to keep the same power per unit area, while increasing throughput through parallelism.

The value of a computing technology can be thought of as the amount of computation performed per dollar spent, which includes the cost of the system plus the power to run it. The amount of computation will be proportional to clock rate, say Df, for a constant D and clock rate f. While the cost of any specific chip or a gate on a chip is fixed at a constant A, there is an expectation that cost per device will continue decline exponentially due to Moore's Law [5]. Therefore, the cost of a device, gate, or memory cell is A exp($-\alpha\, t$), where A is the cost today, α is the rate of Moore's Law for density only, and t is the amount of time into the future. While cost of electricity has been approximately flat over decades, adiabatic circuits consume O(f) energy per operation. If the operations are performed at rate f, energy cost over a fixed lifetime is the product of these last two factors, or Cf^2, for a constant C. Overall value V to the user is

$$V = Df \,/\, (A\, e^{-\alpha\, t} + Cf^2) \qquad (1)$$

Fig. 1 is a plot of this equation, showing the relevant aspects of behavior. We refer to the peak in this plot as a ridge. On the right-hand side of the ridge, the clock is so fast that the Cf^2 cost for power significantly exceeds the cost of the computer. As the clock rate rises, value to the user drops because the logic becomes inefficient in its use of power. Throughput drops on the left-hand side of the ridge due to the clock rate being too slow. While energy is not a concern in this region, the user must purchase a bigger computer with more chips to realize a given amount of throughput. The optimum clock is at the peak of the ridge, which turns out to be where half the user's money goes to buy the computer and the other half to power it. See [6] for a more detailed derivation.

The peak of the ridge rises over time, which will be the basis of a 3D analogy to Moore's Law. If we use adiabatic circuits and tie just the right amount of clock rate reduction to scaling, we stay on the top of the ridge and get a rising economic value

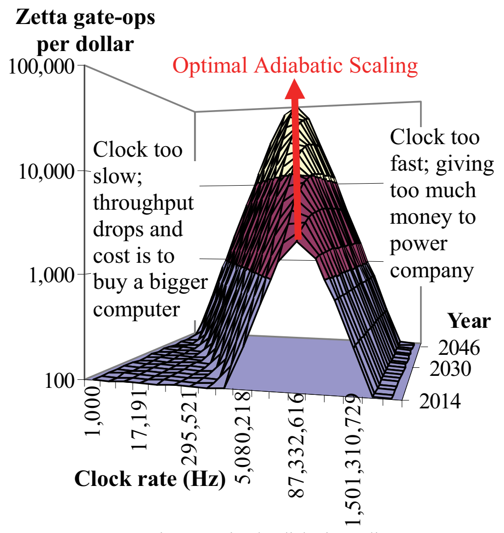


Fig. 1: Optimal Adiabatic Scaling

from our chips. If we ignore clock rate, the user should see that we are likely to fall off the top of the ridge and get less value.

Another way to view the situation is that we are creating computing from two raw materials, devices and energy. Adiabatic circuits allow us to trade off the relative proportions of these two inputs. If one input (devices) declines in cost relative to the other (energy), the economically wise approach is to use more of the cheaper resource so the final product (computing) declines in cost as much as possible.

We can now make a scaling rule through a recurrence relation, although the starting point will not be described until section III. Say we are the manufacturer of a circuit board with a 3D adiabatic chip, as shown in Fig. 2. This circuit board has a socket for the chip, a clock generator at frequency f, and a cooling system adequate for that chip. The chip maker tells us that due to technology improvements, they have a new chip with 4× as many features in the third dimension and hence 4× as many devices. If we simply plug the chip into the socket, it will dissipate 4× as much power and exceed the cooling capacity of the system. However, if we decrease the clock rate by √4× = 2× (i.e., the new clock rate is f/2), then we will have 4 times as many devices each operating at $½^2$ = ¼ of the original power. This will create the same power per unit area on the face of the 3D solid in contact with the heat sink. However, we now have 4× as many devices operating at ½ the clock rate. This raises throughput 2× (defining throughput as the number of gates times clock rate, as is done in the semiconductor industry). If the current system was optimal in the sense of being on the ridge in Fig. 1, the new system will be as well. The starting point of this recurrence relation must be an architecture that can translate this type of scaling into useful computation.

Scaling will be limited by temperature rise due to heat flow in the Z direction, but a brief discussion is warranted to verify this is far off. Fig. 2 shows a heat flow scenario for an exemplary 25 watt chip, partitioned to have 20 watts generated right at the heat sink and 5 watts assumed to be produced on the opposite face. Let us assume a 10°K temperature rise is acceptable in the Z direction across the solid volume of the chip. The solid volume will comprise a mixture of devices, wiring, support, etc. with a thermal conductivity that we can only estimate at this point. If the solid volume of the chip is equivalent to a good heat conductor like Silicon or Aluminum, a simple calculation illustrated in Fig. 2 shows this approach

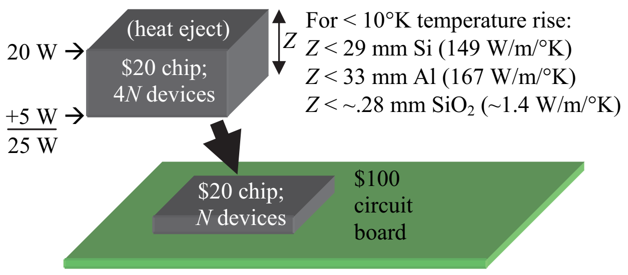

(heat eject)
20 W
$20 chip; 4N devices
Z
+5 W
25 W
For < 10°K temperature rise:
Z < 29 mm Si (149 W/m/°K)
Z < 33 mm Al (167 W/m/°K)
Z < ~.28 mm SiO2 (~1.4 W/m/°K)
$20 chip; N devices
$100 circuit board

Fig. 2 Scaling scenario

will work up to a Z dimension of several centimeters. Several centimeters is much more than needed, so we do not discuss this matter further.

The discussion above can be cast in the terminology of a semiconductor scaling rule. The left side of Table I (with values in red) comprises a subset of the scaling rules documented in [7], whereas the right column contains the corresponding expressions for OAS with natural redefinitions as discussed below.

We need to emphasize some aspects of the standard semiconductor scaling rules to establish a basis of comparison with OAS. Semiconductor scaling is based on reduction of feature size λ, which is the inverse of the number of features per unit distance α. The column *Const field* is the original Dennard scaling. This scaling rule has α^3 in the last row labeled $f \times N_{tran} \times N_{core}$, where N_{tran} is the number of transistors per core and N_{core} is the number of cores per chip. This expression is throughput as defined in the semiconductor industry. The third power on the scaling parameter (α^3) is the rapid scaling of the original Moore's Law, and is stronger than any current scaling rule or any likely in the future.

At the point of final scaling in 2D, voltage V and capacitance per unit area or device C will stop scaling. The only scaling rule in the cited work [7] with this property is in the column labeled *Const* V, f, N_{tran}. The value in the bottom row of this column is 1, meaning no throughput scaling. In other words, once C and V stop scaling, industry appears to be projecting that throughput will no longer rise.

However, it is necessary to redefine some terms in the industry scaling rules that are intrinsically tied to scaling in two dimensions. We make the following adaptations to 3D:

[*] L_{gate} does not change (and is no longer necessary);

[†] $N = \alpha^2$ is defined to be the scaling of the number of features in the third dimension assuming no more scaling in the first two dimensions. N is in lieu of L_{gate}, W, and L_{wire}, which no longer change;

TABLE I
Optimal Adiabatic Scaling as a semiconductor scaling rule

	Const field	Const V, f, N_{tran}	Multi core	OAS (new)
L_{gate}	$1/\alpha$	$1/\alpha$	$1/\alpha$	$L = 1$[*]
W, L_{wire}	$1/\alpha$	1	$1/\alpha$	$N = \alpha^2$[†]
V	$1/\alpha$	1	1	1
C	$1/\alpha$	1	$1/\alpha$	1
$U_{stor} = \frac{1}{2}CV^2$	$1/\alpha^3$	1	$1/\alpha$	$E_g = 1/\sqrt{N} = 1/\alpha$[‡]
F	α	1	1	$1/\sqrt{N} = 1/\alpha$
N_{tran}/core	α^2	1	1	1
N_{core}/A	1	1	α	$N = \alpha^2$
P_{ckt}	$1/\alpha^2$	1	$1/\alpha$	$1/\sqrt{N} = 1/\alpha$
P/A	1	1	1	1[§]
$f \times N_{tran} \times N_{core}$	α^3	1	α	$\sqrt{N} = \alpha$

[‡] The common definition of signal energy, $U_{stor,}$ assumes signal energy is turned into heat after one use. However, adiabatic circuits, perhaps cleverly, use the same signal energy many times before it is turned in to heat. To accommodate, we redefine this entry to be E_g, the heat dissipated on each operation;

[§] We clarify that power is measured from the 2D surface of a 3D structure that is in contact with the heat sink, including power from all the devices that may be stacked on top of the 2D surface.

We can put OAS's throughput growth of $\sqrt{N} = \alpha$ into context. At α^3, the original Dennard scaling was much stronger than current multi core and OAS, which are α. However, the multi-core scaling rule requires continued reductions in C to function, while OAS uses adiabatic circuits and 3D to retain the same rate α even after 2D scaling ends.

III. A Circuit to Start the Recurrence Relation

The previous section presented a scaling rule derived from a recurrence relation, but a recurrence relation needs a starting point. This section describes a suitable adiabatic circuit that will also serve as a building block for a larger architecture. A discussion of the architecture and range of computations it can perform is presented in Section IV.

The circuit shown in Fig. 3 is a variant of a circuit experimentally demonstrated and documented in [8] for Boolean vector-matrix multiply. We will not use the multiply function in this paper, but the circuit employed an adiabatic memory-type access scheme that we will adopt here.

The circuit is a memory that recycles electric charge on the rows. A typical memory comprises banks of some size, such as 1024×1024 (rows×columns), as determined by electrical properties. The memory scales by having more banks, as opposed to larger banks. A typical memory applies a voltage to one row, and then removes it and applies the voltage to a different row. Driving a row, at minimum, consumes CV^2 energy, where C is the capacitance of the row or rows receiving the voltage. In contemporary memories, the CV^2 energy is turned into heat every time a new row is accessed.

The circuit shown has rotator switches that connect one row from each memory bank to a common wire, with an inductor on that wire. This forms what is called a *tank circuit* comprising the inductor, the capacitance of one row from each

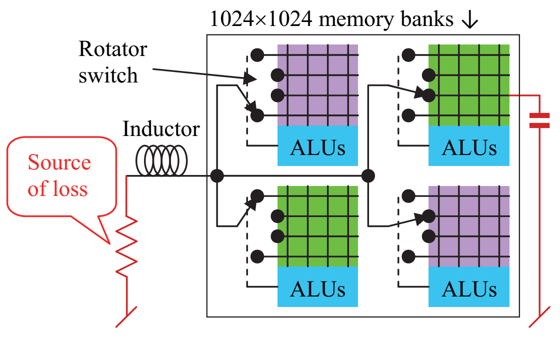


Fig. 3: An adiabatic circuit compatible with OAS

memory bank in parallel, and resistance in the switch and the various wires. This circuit oscillates sinusoidally (i.e, sin(f t)). However, the circuit has another degree of freedom. The position of the rotator switches can be changed. As long as the switches change only when the sine wave goes through a specific voltage, there will be no energy dissipation due to abrupt charging or discharging of the capacitors. This allows a random access pattern to the memory, albeit synchronized.

This circuit has an energy efficiency advantage, which is like a figure of merit. Ref. [8] demonstrated an 85× figure of merit (i. e. the fraction 84/85 of the CV^2 energy was recycled). The figure of merit is due to energy loss in the red components of Fig. 3. A tank circuit does not dissipate CV^2 energy per cycle, but has I^2R losses in the resistor. However, the current will be proportional to operating frequency f, and hence I^2 will be proportional to f^2. This becomes the Cf^2 term in (1).

The specific circuit in [8] has undesireable attributes that deserve mention and that will have implications later on. Most common memory circuits (DRAM, SRAM) have other dissipation mechanisms besides the row capacitance. These other dissipation mechanisms would generally make high figures of merit impossible. The experimental demonstration in [8] used a charge injection device as the memory cell. As a memory cell, this three-transistor device can either be an open circuit or a capacitor. Neither open circuits nor capacitors dissipate energy.

IV. Architecture and Programming

The preceding discussion shows a basis for structures fitting the model in Fig. 3 scaling up over time. However, will they be useful for a broad enough range of applications to justify investment?

The PIMS architecture shown in a potential physical form in Fig 4 was engineered as a compromise between architectural versatility and being a suitable starting point for the scaling rule. The basic structure has a CMOS layer with ALUs and memory drivers in contact with a heat sink dissipating constant power per unit area in accordance with the recurrence relation. The replication unit comprises an ALU plus memory cells layered above the ALU, as shown.

Scaling can take place by increasing the number of layers, or by stacking multiple copies of the structure comprised of both the CMOS and its 3D memory.

We will use sparse matrix-vector multiply (SpMV) $xA = y$ illustrated on the top of Fig. 5 as our example. Unlike the dense linear algebra operations used in the LINPACK benchmark, SpMV has performance limited by memory bandwidth rather than floating-point arithmetic throughput. Many numerical and graph algorithms depend on SpMV, or variations of it. It is also widely used by Deep Learning neural networks in artificial intelligence. A reasonable C implementation takes three lines of code, yet such a simple implementation results in an irregular and inefficient memory access pattern. We assumed that algorithms of this type would

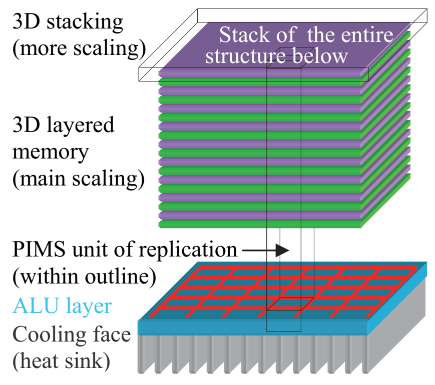

3D stacking (more scaling)
Stack of the entire structure below
3D layered memory (main scaling)
PIMS unit of replication (within outline)
ALU layer
Cooling face (heat sink)

Fig 4. PIMS architecture

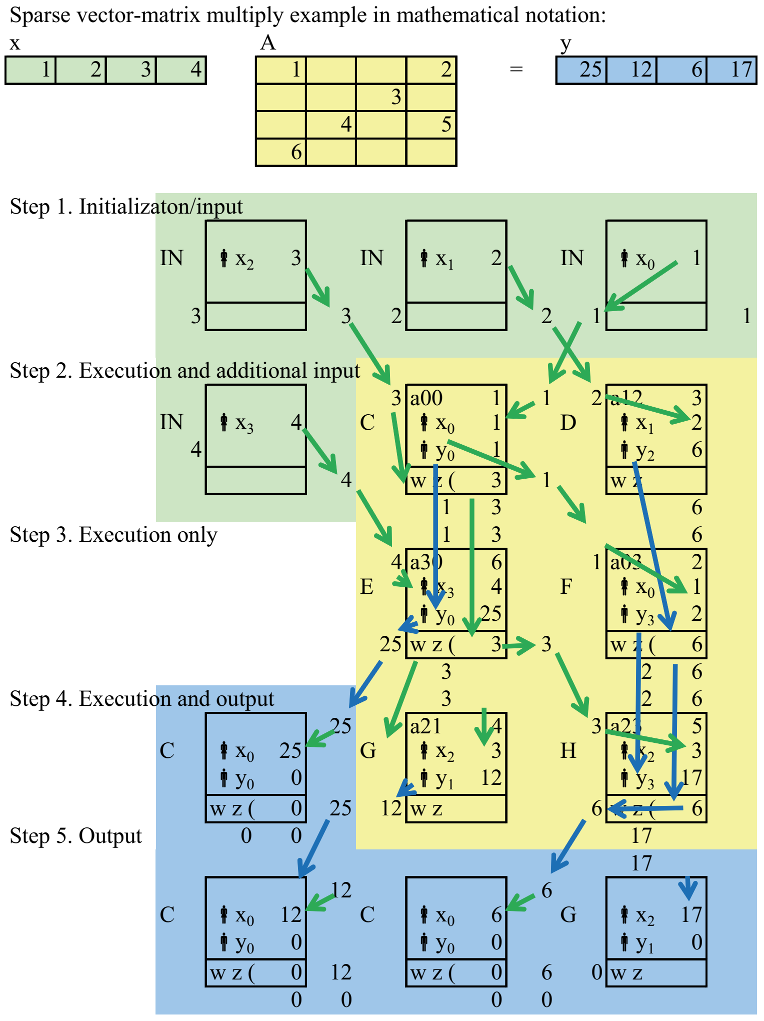

Sparse vector-matrix multiply example in mathematical notation:
x
A
y
Step 1. Initializaton/input
Step 2. Execution and additional input
Step 3. Execution only
Step 4. Execution and output
Step 5. Output
IN
C
D
E
F
G
H
a00
a12
a30
a03
a21
a23
w z (

Fig. 5: PIMS programming example

be memory bound even for PIMS. In other words, even with the best adiabatic memory we could hope for, the energy to access a bit from memory would exceed the energy required to process the bit. This insight justified our proposing an architecture with an innovative memory coupled to ordinary CMOS for arithmetic.

We found the key to successful PIMS programming was to make the memory access pattern as regular and as local as possible. In fact, the example in Fig. 5 accesses memory in one sweep from top to bottom (although other algorithms require more elaborate patterns). The central concept to sparse vector-matrix multiply is to precompute the layout pattern of a sparse matrix so it can be processed in regular order.

The top of Fig. 5 shows the vector and sparse matrix in standard mathematical notation, while the bottom shows how they would be allocated to PIMS memory in the circuit of Fig. 3 or architecture of Fig. 4. One should imagine PIMS memory in Fig. 4 to have five rows, divided into groups of columns of some word width (word width is arbitrary, although we use 16-bit integers in the example that follows). We also labeled the rows of the diagram by steps. If the memory is accessed in a uniform pattern from to top to bottom, the steps and row addresses will match up.

Fig. 5 is from an Excel spreadsheet where a matrix-vector multiply in the space-inefficient form at the top was rearranged into the packed form at the bottom, in a manual activity equivalent to precomputing an efficient layout pattern. This paper only shows the spreadsheet as an image, but the actual spreadsheet is available in [6]. If a reader wants reassurance that the algorithm works, he or she would be able to type over any nonzero input value in the spreadsheet and see the output value change.

While systolic arrays have been known for years to be capable of doing dense vector-matrix multiply, sparse matrix operations ordinarily require an irregular access pattern to avoid unnecessary computing on zero matrix entries. This is reflected in Fig. 5 by green and blue arrows, which represent the input and output vectors jumping between memory elements. A reader can easily see that the data movement arrows all go downwards and no further than a nearest neighbor cell horizontally. The reader can check and see that all the arithmetic operations needed for the specific SpMV problem are actually performed if data is moved as indicated by the arrows.

Fig. 6 illustrates the logic for each ALU in Fig. 3. For SpMV, each ALU needs a multiplier and an adder, which are illustrated with the corresponding symbols × and + in Fig. 6. In addition, each ALU needs to move intermediate results to neighboring ALUs or store them temporarily. This data movement corresponds in Fig. 6 to the wait zone path, and to the input and output paths on the top and bottom of the logic diagram The overall operation for SpMV is that the memory in Fig. 4 accesses the five rows in Fig. 5 in five steps. This simple example requires three copies of the ALU in Fig. 6, which receives the memory output in the box labeled "mem." The memory also contains an opcode, for which this example requires 7 opcodes. The opcodes control the way the ALUs

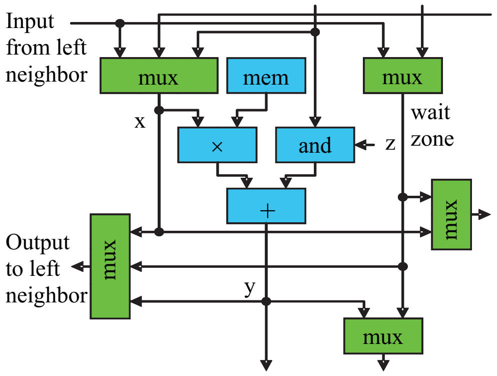


Fig. 6: Example ALU layout

shift data in order to create the data flow pattern of the blue and green arrows in Fig. 5 (see [6] for the actual opcodes). Also, the input and output vectors are stored in memory in the green and blue areas of Fig. 5 (also detailed in [6]).

V. Quantitative Analysis

While PIMS has been presented as a scaling rule, people actually build specific systems and the scaling rule emerges over time. We now provide a quantitative estimate of the performance boost of systems that could be from two generations. Due to popularity of neural network research, we use SpMV as a test problem where the vector length is 10^{11} and the matrix has a total of 10^{15} out of a potential 10^{22} nonzero matrix elements (these are the neuron and synapse figures for a human brain). The arithmetic is 16-bit fixed point. We analyzed power requirements for both logic and memory components separately, creating a grid of the combinations in Table II.

We analyzed three options for memory: (1) a consumer-grade nVIDIA GTX 750 Ti GPU (which is memory bandwidth limited for this application); (2) a hypothetical DRAM-based PIM based on a Micron DRAM; and (3) an adiabatic memory based on [8]. These solutions have access energy E_{bit} of 100,000, 46, and .9 fJ/bit.

Table II includes two logic options from [9]: HomJTFET and CMOS HP. CMOS HP is the current pre-eminent logic technology. However, the MOSFET underlying CMOS is unable to scale supply voltage. There is intense research on

TABLE II
Simple performance assessment

Memory (E_{bit} for reference)	GTX 750 Ti	DRAM	Adiabatic memory
	.1 nJ/bit	46 fJ/bit	.9 fJ/bit
Logic type			
HomJTFET			
7.7 fJ/mult	43.4 MW	23.2 KW	607 W
		.7 % logic	25 % logic
CMOS HP			
127.8 fJ/mult	43.4 MW	25.6 KW	3010 W
		10 % logic	85 % logic

what are essentially drop-in replacements for transistors where voltage can scale, TFET being the most popular example. There will be more discussion of the logic below.

Compared to today's technology (GTX 750 Ti GPU), the proposed PIMS approach reduces the power requirements for our SpMV example application from 43.4 MW to 607 W, for an energy efficiency improvement of 71,000×. The 71,000× includes 2000× due to architectural changes and up to 35× due to the adiabatic scaling. However, the scaling rule should permit the 35× factor to grow whereas the architectural factor is less likely to grow.

We stated earlier that memory energy predominates over logic energy. The numbers in red in Table II are the percentage of the system energy consumed by logic, of which only three support this assertion. When regular CMOS is used for logic (CMOS HP), the adiabatic memory offers such a large improvement in energy efficiency that the CMOS ends up consuming 85% of the system power. This clearly calls for further discussion. Industry needs an improvement for the MOSFET to continue scaling, but improved transistors are not a topic of this paper.

VI. PIMS Programming Generalization

The previous section presented one example algorithm. Besides sparse matrices, we are studying sorting, text search, hashing, probabilistic context-free grammar parsing, LINPACK, and database functions like indexed storage with transaction/rollback.

We also studied the possibility of a computer with OAS but where the clock rate could be varied on the fly by region. For example, 99% of the ALUs could be effectively powered off by running them at f=0 while the remaining 1% execute at 10f (say f is normal speed). This 10× boost actually results in 10× higher energy per operation and 10× more operations per second, for a 100× power increase on 1% of the processors. In other words, total power remains the same. This approach is consistent with OAS, yet would satisfy the need for a few fast threads.

VII. Conclusions

This work presents a new approach to computing that is grounded in current technologies, but changes several of the underlying technologies in concert to achieve a new goal. In particular, this work accomplishes the following:

1. Exploits adiabatic circuits for memory as opposed to logic.
2. Enables an increase in energy efficiency through adiabatic circuit techniques applied to *unchanging devices*, which is not a property of current scaling rules.
3. Formulates a scaling rule that follows an improvement path as a function of industry's ability to add features in the third dimension (layers).
4. Proposes the PIMS architecture from memory structures that are compatible with OAS. The von Neumann architecture is incompatible with OAS.
5. Performed a basic analysis of the degree of power efficiency improvement that could be expected. While the physics of computation suggest adiabatic computation can have arbitrarily low energy consumption (not counting irreversible logic operations), we used an 85× energy reduction from a experimental demonstration. With these figures we obtained a rough guess of 71,000× energy efficiency improvement when combining both architecture and circuit benefits..
6. A PIMS architecture that can switch between high thread speed with low parallelism to massive parallelism with low thread speed. PIMS can actually vary the energy efficiency of gates (J/op) on the fly, which is unprecedented in our experience.

One direction of future work is to further characterize the range of applications compatible with this model. We have done some of this, but not reported here due to space limitations. There is more to do.

A second direction is to understand the limits of scaling. Other approaches that compete with PIMS often use devices that exist only in concept. PIMS has the advantage that the devices needed for a first version are already in production. However, if the approach in this paper becomes widely used, there will be refinements to the device. It would be interesting to know how many generations of PIMS scaling will be possible.

A third direction is to further explore nanotechnology technologies for adiabatic memory. Ref. [8] experimentally demonstrated an adiabatic memory based on a call containing three transistors organized as a charge-injection device. This has advantages over DRAM (which dissipates CV^2 energy to ground) and RRAM/PCM which are inherently resistive. There may be a more practical memory cell that is lossless for reading – for example a memcapacitor.

A fourth direction is to address energy efficiency of logic. If memory energy is adequately reduced, logic energy becomes the next challenge.

References

[1] Y. Fujisaki, "Review of emerging new solid-State non-volatile memories," Jpn. J. Appl. Phys. 52 040001.

[2] C. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development*, Volume 17 Issue 6, November 1973, Pages 525-532

[3] RP Feynman, JG Hey, RW Allen, *Feynman Lectures on Computation*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998

[4] Koller, J.G.; Athas, W.C., "Adiabatic switching, low energy computing, and the physics of storing and erasing information," *Proceedings of Physics of Computation Workshop. 1992.*

[5] G. Moore, "Craming more components onto integrated circuits," *Electronics*, pp 114-117, April 19, 1965.

[6] E DeBenedictis, "Scaling to nanotechnology limits with the PIMS computer architecture and a new scaling rule," Sandia technical report SAND2015-1318.

[7] Theis, Thomas N., and Paul M. Solomon. "In quest of the 'next switch': prospects for greatly reduced power dissipation in a successor to the silicon field-effect transistor." *Proceedings of the IEEE* 98.12 (2010): 2005-2014.

[8] R. Karakiewicz, R. Genov, G. Cauwenberghs, "1.1 TMACS/mW fine-grained stochastic resonant charge-recycling array processor," *IEEE Sensors Journal*, Vol. 12, No. 4, April 2012, p. 785.

[9] D. Nikonov and I. Young. "Overview of beyond-CMOS devices and a uniform methodology for their nenchmarking", *arXiv* 1302.0244.

Energy Efficiency Limits of Logic and Memory

Sapan Agarwal[1], Jeanine Cook[2*], Erik DeBenedictis[2], Michael P. Frank[2]
[1]Microsystems Science and Technology
[2]Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
[*]corresponding author jeacook@sandia.gov

Gert Cauwenberghs[3]; Sriseshan Srikanth[4], Bobin Deng[4], Eric R. Hein[4], Paul G. Rabbat[4], Thomas M. Conte[4];
[3]Department of Bioengineering, Jacobs School of Engineering, and Institute for Neural Computation, University of California San Diego, La Jolla, CA,
[4] Schools of CS and ECE
Georgia Institute of Technology, Atlanta GA

***Abstract*—We address practical limits of energy efficiency scaling for logic and memory. Scaling of logic will end with unreliable operation, making computers probabilistic as a side effect. The errors can be corrected or tolerated, but overhead will increase with further scaling. We address the tradeoff between scaling and error correction that yields minimum energy per operation, finding new error correction methods with energy consumption limits about 2× below current approaches. The maximum energy efficiency for memory depends on several other factors. Adiabatic and reversible methods applied to logic have promise, but overheads have precluded practical use. However, the regular array structure of memory arrays tends to reduce overhead and makes adiabatic memory a viable option. This paper reports an adiabatic memory that has been tested at about 85× improvement over standard designs for energy efficiency. Combining these approaches could set energy efficiency expectations for processor-in-memory computing systems.**

Keywords—Moore's Law, Shannon, Landauer, limits of computing, adiabatic, reversible, reversible logic, millivolt switch

I. INTRODUCTION

We address an apparently novel tradeoff between two well-known issues. Semiconductor scaling is widely known to reduce energy consumption today, but it will eventually lead to a rise in errors due to insufficient energy to distinguish between 0s and 1s. Algorithm-Based Fault Tolerance (ABFT) and error correction are well-known methods that allow logic and memory to function in the presence errors, albeit with progressively more overhead as the error rate rises. We discuss how continued scaling will initially reduce energy consumption, but scaling beyond an optimal point will cause energy to increase again due to the overhead of handling the errors. This paper finds two error correction methods that reduce minimum energy consumption for logic, yet finds a different approach that is more suitable to memory.

When Moore's Law was formulated in the 1960s [1], it projected practical, manufacturable semiconductor technology from that point in time into the future. In the same decade, theorists identified energy efficiency limits for computer technology [2] that were unimaginably far ahead of the technology of the day. The big gap led to 50 years of exponential growth. Now in the 2010s, we find the energy efficiency of manufactured devices being in the range of 10× to 10,000× above the theoretical limits.

Device size is approaching theoretical limits as well. However, the expected change from 2D to 3D manufacturing will allow module-level density to rise and further exacerbate energy efficiency challenges.

Each 2× scaling generation enables new products for a few years in the huge global information technologies (IT) sector. Given the stakes, it would be useful to find the endpoint of scaling more accurately than just 10×-10,000× beyond where we are now.

The best known work on minimum dissipation is due to Landauer [2], who stated that the minimum energy is typically on the order of kT for each irreversible logic operation. The expression $kT \approx 4\times10^{-21}$ joules at room temperature comprises Boltzmann's constant k times the absolute temperature T. Landauer's minimum is a very solid lower bound yet often misinterpreted; another paper at this conference analyzes this issue [3].

Today's Complementary Metal-Oxide Semiconductor (CMOS) can never reach Landauer's minimum, but it is important to know how close it can come. CMOS is a term that denotes both a complementary pull-up/pull-down logic circuit and the MOSFET transistor. The CMOS circuit design is very simple, but the circuit's simplicity forces the charging and discharging of capacitors defining signal values directly from DC power supplies. The simple charging circuit limits energy efficiency to what is called the Landauer-Shannon limit [4] that is about 50× higher than Landauer's minimum.

However, the transistors have issues as well. Roadmaps for transistors project additional reduction in power supply voltage for a device class called millivolt switches. This phrase refers to devices that could operate in CMOS-like circuits below the limits of MOSFET devices. MOSFETs are limited to $\ln(10)\,kT/q \approx 60$ mV/decade sub threshold slope, which prevents practical operation below about 0.5 V. While a specific MOSFET replacement with steeper slope has not been selected for full-scale development, Tunnel FETs and

Approved for unclassified unlimited release SAND2016-7245 C. Research supported by Sandia National Laboratories, a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000

Piezotronic FETs are candidates. If scaling continues after these devices go into production, technology should be able to reach the Landauer-Shannon limit (and we argue it can go further).

The theoretical minimum energy for CMOS Boolean logic has been studied by considering the wires between logic gates as communications links and applying Shannon's communications theory, but we will argue this does not find the true minimum. Just as a cell phone picks up more and more static as it moves further from the transmitter tower, the theoretical minimum energy of a Boolean Logic gate is not a single number but a function of the acceptable error probability. The earliest analysis the authors can find [4] yields the familiar expression $P_e = \exp(-E_{\text{signal}} / kT)$ [1]. Modern textbooks [5, p. 595] sometimes use the asymptotically equivalent complementary error function (erfc). The erfc version appears in [6] as applicable to minimum energy for CMOS, however Theis and Solomon follow their analysis with the statement "[a]s thermal voltage fluctuations become significant, we must incorporate redundancy and error correction in the logic to keep the error rate in bounds" [6].

There is about a 50× "end zone" to energy scaling that this paper begins to address with error correction. CMOS circuits made of extremely good millivolt switches should be able to reach the Landauer-Shannon limit of $P_e = \exp(-E_{\text{signal}} / kT)$, but clever circuits will be needed to get closer to Landauer's minimum of kT. The historical literature contains (unimplemented) examples of designs [4] [7], supporting the idea that such approaches are possible and we find some here.

There are methods of correcting gate errors without concern to energy consumption. For example, Triple Modular Redundancy (TMR) [8] votes the results of three redundant calculations and uses the winner as the answer. This is effective, but more than triples the gate count and hence energy. Another class of techniques is called Algorithm-Based Fault Tolerance (ABFT) [9], which typically performs a full calculation and an estimate. For example, the estimate might be just the least significant bit of the full computation. The calculation is repeated if the two do not match. ABFT may have low overhead, but the versions in the literature are specific to just one algorithm rather than applying generally.

To the knowledge of the authors, this is the first paper to connect the exponential error probability $P_e = \exp(-E_{\text{signal}} / kT)$ with error correction overhead. For example, cutting signal energy in half ($E_{\text{signal}}' = E_{\text{signal}}/2$) will cut gate energy in half but change $P_e' = \sqrt{P_e}$. If error correction can square the error probability for less than a factor of two in overhead, the energy efficiency can rise above the Landauer-Shannon limit.

Reversible computing [10] is a second approach to beating energy efficiency limits, yet it faced practical limitations in the past. Reversible computing can use existing MOSFET transistors, yet uses different circuits that recycle energy. Readers are referred to [11] for extensive additional details, but the principle for energy efficiency scaling is to recycle a rising fraction of the energy as the technology improves. One generation might recycle 99% of operating energy, drawing only the remaining 1% from the power supply. A few generations later, 99.9% might be recycled with 0.1% drawn from the power supply. And so forth.

There have been challenges in applying reversible computing to logic, but there is progress in the area including a paper at this conference by Snider et. al. [12]. The circuitry needed to recycle energy is more complicated than conventional Boolean logic, using more transistors in some approaches and using large numbers of clock signals in others. The cost to power a processor over its lifetime has recently started to exceed the purchase cost, making more complexity a good investment if it can lower energy consumption.

Reversible computing principles can be applied to memory as well, which is being reported in this paper. Complexity has different meanings for memory and logic. Memories are divided into addressing logic and a memory storage array. While smaller storage cells are preferred, the user wants the largest possible array for a given storage cell size. This is because the array holds the data that is valuable to the user. As long as the addressing logic is small compared to the memory array, the user will not care about its complexity. Logic associated with memory addressing is a large consumer of energy in current computer systems, so making the addressing more energy efficient is a priority as long as it does not increase complexity very much.

In section IV, we report on a memory using adiabatic principles that has been measured as reducing energy by 85× (i. e. recovers a fraction 84/85 of the delivered energy drawn from the power supply) owing to resonant energy exchange.

II. Redundant Residue Number Systems

This section shows that an RRNS-based processor can exceed the energy efficiency predicted by the Landauer-Shannon limit. A companion paper in these same proceedings [13] describes the Computationally-Redundant Energy-Efficient Processing for Y'all (CREEPY) architecture. CREEPY uses a n=4 sub cores to represent ~32-bit numbers using a Residue Number System (RNS), with one residue per sub core. CREEPY also has r=2 additional redundant sub cores that extend the RNS into a Redundant RNS (RRNS) and allow detection and correction of a single logic error. The RRNS was developed in [14], but that paper did not consider energy efficiency.

CREEPY (or any RRNS processor) can be used as a baseline for comparisons by ignoring both the energy consumption of the redundant sub cores and their ability to correct errors, a method developed in [15] for general circuits. The energy efficiency of the baseline can be improved by reducing E_{signal} and increasing the energy efficiency of the underlying gates up to the Landauer-Shannon limit in [4] and [5, p. 595], as described above. Any single error that occurs with probability $P_e = \exp(-E_{\text{signal}} / kT)$ per gate operation would cause a system failure. However, [6] suggests that incorporation of redundancy and error correction might be helpful to further increase energy efficiency.

[1] The Landauer-Shannon limit is usually written as $E_{\text{signal}} = kT \ln(1/P_e)$

CREEPY can then model redundancy and error correction by including the energy consumption of the redundant sub cores and assuming they will correct single errors. In this case, a system failure occurs only when two or more errors occur within a time window. The analysis below shows that redundancy and error correction can help. If E_{signal} is lowered the precise amount that keeps overall system energy unchanged given the additional gates, the probability of a system failure declines—at least in some useful operating ranges. If the baseline was operating at the Landauer-Shannon limit, the RRNS version would operate below the limit.

Consider the baseline system where each of n residues is implemented by G gates, so the baseline comprises $N = Gn$ gates. While the baseline does not check or correct errors, we will derive the error probability on batches of f_{n} sequential arithmetic operations, using notation consistent with [13]. Since all errors will be undetected, the probability of an undetected error per batch as a function of E_{signal} is

$$P_{\text{u}}(E_{\text{signal}}) = Nf_{\text{n}} \exp(-E_{\text{signal}} / kT). \qquad (1)$$

Now consider the additional r redundant residues for a total of $t = n+r$ residues, and an additional $R = Gr$ gates. This RRNS circuit will be distinguished by primes (') and operated with a signal energy E_{signal}'. The probability of an undetectable double error in a batch will be

$$P_{\text{u}}'(E_{\text{signal}}') = ½\, t(t-1)\, (G f_{\text{n}})^2 \exp(-2E_{\text{signal}}' / kT), \qquad (2)$$

which are the ½ $t(t-1)$ combinations of two residues being in error multiplied by the square of the probability of each residue being in error over the time of an entire batch. We are not detecting or correcting errors at this point; errors just become inconsistent encodings of the RRNS residues.

Assume the single error detection and correction is performed once for each batch of f_{n} operations by a circuit comprising C gates. We will not model the gates explicitly, so C will be an equivalent value.

The two circuits will consume the same total energy if we set

$$E_{\text{signal}}' = N / (N+R+C/f_{\text{n}})\, E_{\text{signal}}. \qquad (3)$$

The ratio of the two error probabilities above form a figure of merit $\mathcal{M}$ if the probabilities are computed at constant total energy, which can be the result of adjusting the signal energies of the two circuits to match as in (3)

$$\mathcal{M} = \frac{P_{\text{u}}(E_{\text{signal}}).}{P_{\text{u}}'(E_{\text{signal}}')} \qquad (4)$$

The figure of merit can be expressed either as a function of E_{signal}' or E_{signal}. We choose E_{signal}. If we designate $\beta = ½\, t(t-1)/n$ as an RRNS property and $G^* = N + R + C/f_{\text{n}}$, (4) becomes

$$\mathcal{M} = \frac{Nf_{\text{n}} \exp(-E_{\text{signal}} / kT)}{\beta\, n(G f_{\text{n}})^2 \exp(-2N/G^*\, E_{\text{signal}} / kT)}. \qquad (5)$$

Now (5) simplifies to

$$\mathcal{M} = 1/(\beta\ \ Gf_{\text{n}}) \exp\left((2N-G^*)/G^*\, E_{\text{signal}} / kT\right). \qquad (6)$$

The error detection yields benefit when $\mathcal{M}$ is greater than 1, or beyond the break even point. Shifting to the inequality and taking the logarithm of both sides yields

$$0 < \ln(1/(\beta\ \ Gf_{\text{n}})) + (2N-G^*)/G^*\, E_{\text{signal}} / kT, \qquad (7)$$

which can be solved for signal energy in units of kT as

$$E_{\text{signal}} / kT > \ln(\beta\ \ Gf_{\text{n}})\, G^*/(2N-G^*), \text{ or} \qquad (8)$$

$$E_{\text{signal}}' / kT > \ln(\beta\ \ Gf_{\text{n}})\, N/(2N-G^*). \qquad (9)$$

Let us explore the range of situations where RRNS is helpful in raising energy efficiency. Assume the number of logic gates in a residue calculation is $G = 2000$, a number used in [13]. The example number system from [14] used in [13] uses n=4, r=2, and therefore β = 3.75, $N = Gn = 8{,}000$, and $R = Gr = 4{,}000$. From an inspection of diagrams in [14], let us assume detection and correction is equivalent to 3 arithmetic operations or 12 residue operations. This implies $C = 12G = 24{,}000$.

The spreadsheet in Table I shows RRNS can raise energy efficiency as long as the signal energies are above the break even point. As long as f_{n} is more than 10 or so, the break even signal energies are below anything useful in a design, so the break even point is not an obstacle. Even though Table I establishes the boundary where energy efficiency rises, the specific numbers in Table 1 are the point where RRNS makes precisely no difference.

Based on the f_n = 100 reliability analysis in the companion paper [13], we conclude RRNS gives benefit in at least one

Parameters				Break even	
f_{n}	N	R	C	e_{signal} / kT	e_{signal}' / kT
7	8,000	4,000	24,000	293.45	152.16
8	8,000	4,000	24,000	165.03	88.02
9	8,000	4,000	24,000	122.32	66.72
10	8,000	4,000	24,000	101.03	56.13
11	8,000	4,000	24,000	88.30	49.81
12	8,000	4,000	24,000	79.85	45.63
25	8,000	4,000	24,000	51.76	31.95
50	8,000	4,000	24,000	45.50	29.17
100	8,000	4,000	24,000	44.04	28.78
G	n	r	const	$\exists$	
2000	4	2	12	3.75	

TABLE I. RRNS BREAK EVEN

example situation. The scenario above as analyzed in [13] shows a sharp increase in reliability between E_{signal}' = 42 and 43 kT. Table I shows the break even point for $f_n = 100$ to be E_{signal}' = 28.78 kT, which is lower.

A short discussion may be in order on how to extend the approach. The Shannon-Landauer "limit" has merit, yet we found an exploitable property. Scaling causes a linear reduction in energy consumption but an exponential rise in raw error rate. We searched for a low-overhead error-correction approach and checked to see if there could be a net savings in energy before the exponential dominated. The steepness of the exponential was crucial to how far we could push, e. g. we should have been able to push further with a gentler polynomial but a step function would have been impenetrable. Simply reducing the supply voltage to a semiconductor circuit causes it to fail uncontrollably, which is like a step function. Thus the onset of thermal noise is a special case not expected to be seen before the end of scaling. The authors have no idea how far this could go; it may be the equivalent of one semiconductor generation.

III. Temporal Error Correction

The authors propose a second form of error correction using samples collected over time, which also reduces minimum energy below the Landauer-Shannon limit. Typically, the wire from one gate's output to another's input is modeled as a communications channel. This is shown in Fig. 1A where a wire connects a driving gate through a hypothetical switch to the combined capacitance C of the wire and the next gate's input [6]. The capacitance is charged and then the switch opened. The disconnection leaves a reset noise of voltage kT/C on the input of the gate being analyzed, meeting the Landauer-Shannon limit of $P_e = \exp(-E_{\text{signal}} / kT)$ on the gate's output.

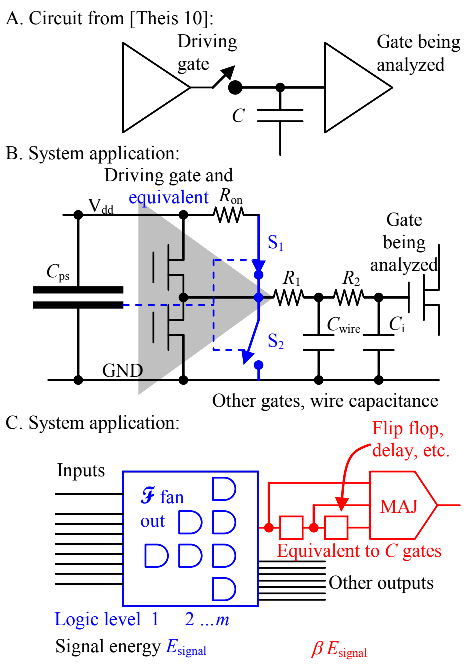


Fig. 1. Temporal error correction

Now imagine that the switch in Fig. 1A is thrown back and forth several times. Since the capacitance would charge to its proper value on the first cycle, later cycles would not consume additional energy. However, the analysis in [6] would be independently valid on each cycle, yielding the same data value but a different sample of the random reset noise. Applying error correction to multiple samples could reduce the error probability—or keep the error probability unchanged while reducing the gate energy. We show an error correction method below that allows more energy to be saved in the logic than is consumed by error correction. The switch is extraneous; it can be left closed all the time or replaced by a wire.

This does not violate the communications theory analysis as the wire between stages will also conduct noise from the receiver's gate backwards though the wire and ultimately to the output gate's power supply. The equivalent circuit model is shown in Fig. 1B in enough detail to convey the basic idea. If R_{on}, R_1, and R_2 are large, it should be clear that the input of the gate being analyzed will have noise voltage kT/C_i. If R_{on}, R_1, and R_2 are small or shorts, they connect the three capacitors C_{ps}, C_{wire}, and C_i in parallel and the noise voltage drops to $kT / (C_{\text{ps}}+C_{\text{wire}}+C_i)$. The R's will never be 0 or ∞ ohms in practice, but this effect seems not to have been considered before.

This type of error correction will need to be applied to multiple levels of logic to amortize the overhead of the error correction circuit. Fig. 1C shows this method being applied to a field of gates in blue organized as a rectangle with m logic levels and n rows, each row comprising an input gate, $m-2$ intermediate gates, and an output. We assume the signal from any given input will affect more than one output, fanning out by a factor $\mathcal{F}$ as it goes from one logic level to the next. Similarly, an output will be controlled by multiple inputs, with a fan-in factor of $\mathcal{F}$ at each level. Each of the n outputs will need to have its own error correction circuit and so we will consider the energy of one row at a time. Data applied to the inputs is transformed to output data in m time steps equal to the propagation delay of a gate, with no error correction between logic levels.

Fig. 1C represents both the baseline and error corrected cases. The baseline circuit comprises only the blue graphics, which are operated at signal energy E_{signal}, using the same terminology as section II. The error corrected case includes the blue graphics operating at a reduced signal energy E_{signal}' followed by the error correction circuitry in red operating at the original signal energy E_{signal}. We define $E_{\text{signal}} = \beta\, E_{\text{signal}}$'.

The example in Fig. 1C uses majority voting over three samples for error correction, but the analysis below uses majority voting of α samples, α ≥ 3 and odd. The inputs must remain stable for least α steps so the outputs can be sampled at the end of steps m, m+1…m+α−1. The propagation delay time is approximately the same as the minimum time needed to get independent error samples. The samples could also be taken by using a single device with a low-pass filter (large capacitance) that averages the signal over multiple propagation delay times.

A. *Error rate P_u of a row of the baseline circuit*

For both error rate and energy calculations, let thermal noise cause a gate to generate an erroneous signal with probability p. The signal will then propagate to the right, spreading to additional rows by a factor of $\mathcal{F}$ each time it moves from one layer to the next. This means the uncorrected error rate from the rightmost blue gate will be the probability of error in a given gate times number of parent gates, $\mathcal{N}$, that fan in to an output:

$$p_{\text{raw}}(p) = p\,\mathcal{N} \tag{11}$$

where:

$$\mathcal{N} = (\mathcal{F}^{m-1} + \ldots\ \mathcal{F}^2 + \mathcal{F} + 1) = (\mathcal{F}^m - 1)/(\mathcal{F} - 1). \tag{12}$$

For the baseline error calculation with no error correction, $p = \exp(-E_{\text{signal}} / kT)$ and

$$P_{\text{u}}(E_{\text{signal}}) = \mathcal{N} \exp(-E_{\text{signal}} / kT) \tag{13}$$

B. *Error rate of a row of the error-corrected circuit (P_u')*

For the calculation with error correction, an undetected error results when a majority of the α samples of the blue logic circuitry are erroneous. However, an undetected error could also result from an error in the red error correction circuit itself. We will call this probability q. The probability of an undetected error is then:

$$\begin{aligned} P_{\text{u}}'(E_{\text{signal}}') &= \sum_{x=\frac{\alpha+1}{2}}^{\alpha} \mathrm{C}(\alpha,x) \times \left(e^{-\frac{E_{\text{signal}}'}{kT}} \mathcal{N} \right)^{x} \times \left(1 - e^{-\frac{E_{\text{signal}}'}{kT}} \mathcal{N} \right)^{1-x} + q \\ &\approx \mathrm{C}\left(\alpha, \frac{\alpha+1}{2}\right) \times \left(e^{-\frac{E_{\text{signal}}'}{kT}} \mathcal{N} \right)^{\frac{\alpha+1}{2}} + q \end{aligned} \tag{14}$$

where $\mathrm{C}(\alpha,x)$ is the number of ways to choose x of the α samples.

The error correction circuit is shown in red in Fig. 1, comprises of two latches and a majority gate. Let us model the circuit as C gates per sample α, where the energy per gate is $\beta E_{signal}'$. The probability of an error occurring in the correction circuit itself is:

$$q = C\alpha \times e^{\frac{-\beta E_{\text{signal}}'}{kT}} \tag{15}$$

So the undetected error rate is

$$P_{\text{u}}'(E_{\text{signal}}') \approx \mathrm{C}\left(\alpha, \frac{\alpha+1}{2}\right) \times \left(e^{-\frac{E_{\text{signal}}'}{kT}} \mathcal{N} \right)^{\frac{\alpha+1}{2}} + C\alpha \times e^{\frac{-\beta E_{\text{signal}}'}{kT}} \tag{16}$$

We can find the error corrected signal energy, E_{signal}', required to get the same error rate as the baseline circuit by setting (13) equal to (16):

$$E_{\text{signal}}' \approx \frac{2}{\alpha+1} E_{\text{signal}} + kT \frac{\alpha-1}{\alpha+1} \ln(\mathcal{N}) + kT \frac{2}{\alpha+1} \ln\left[\mathrm{C}\left(\alpha, \frac{\alpha+1}{2}\right) \right] \tag{17}$$

Here we assumed the number of error correction gates is much less than the number of gates in the logic, $C\alpha \ll \mathcal{N}$, and simplifed the result.

C. *Energy E' of a row of the error corrected circuit*

To calculate the total energy used in a row of error-corrected logic, we need to add the energy consumption of the baseline circuit (blue only in Fig. 1) to compute the correct result, E_0', the energy of the error correction circuit, E_C', and the energy due to errors, E_x'.

The energy of the logic is given by the signal energy E_{signal}' times the number of gates in a row:

$$E_0' = m\, E_{\text{signal}}'. \tag{18}$$

The energy for the error correction circuit is the number of error correction gates times $\beta E_{\text{signal}}'$:

$$E_C' = C \times \alpha \times \beta \times E_{\text{signal}}' \tag{19}$$

Next, consider the energy drawn from the power supply by an ideal CMOS circuit due to thermal errors, E_x' in a single row. Let us assume a thermal error occurs with probability p. Errors in gates on the left side of the network will each propagate to $\mathcal{F}$ rows as they go from layer to layer until they reach level m. An error takes twice the signal energy (an error signal and then a return signal) from the power supply at each level. The number of errors per logic level in one row of the blue gates in Fig. 1C will be:

$$p,\ p(\mathcal{F}+1),\ p(\mathcal{F}^2+\mathcal{F}+1),\ \ldots\ p(\mathcal{F}^{m-1}+\mathcal{F}^{m-2}\ldots 1). \tag{20}$$

The series above can be summed and simplified, yielding an expression for the number of errors in the gates of Fig. 1C per row.

$$N_{\text{err}} = p\,\mathcal{E} = p\,[(\mathcal{F}^m - 1)/(\mathcal{F} - 1)^2 - (m+1)/(\mathcal{F} - 1)], \tag{21}$$

which defines $\mathcal{E}$ as a constant related only to logical circuit structure. The energy due to errors is given by:

$$E_x' = 2\,\alpha\, p\, \mathcal{E}\, E_{signal}' \tag{22}$$

where 2α originates from 2 signal transitions per error times α samples. Thus the total switching energy is:

$$E' = E_0' + E_C' + E_x' = (m + C\times\alpha\times\beta + 2\,\alpha\, p\, \mathcal{E})\times E_{\text{signal}}' \quad (23)$$

D. *Energy of a row of the baseline circuit (E)*

The energy consumption of the baseline circuit (blue only in Fig. 1) will be the signal energy to compute the correct result, E_0, assuming no errors plus the energy, E_x, due to errors. Clearly,

$$E_0 = m\, E_{\text{signal}} \quad (24)$$

and in the baseline case, $p = \exp(-E_{\text{signal}} / kT)$, so

$$E_x = E_{\text{signal}} \exp(-E_{\text{signal}} / kT)\, \mathcal{E} \quad (25)$$

with the sum of equations (24) and (25) being the energy of the baseline, uncorrected, circuit.

Fig. 2 plots energies for circuit representative of a 16×16 bit multiplier with three samples for error correction. The circuit is modeled by m=48 layers of logic with $\mathcal{F} = 32^{1/m}$ such that an error propagates to at most 32 outputs (which is all the outputs that exist on the multiplier). The error correction circuit is modeled by C=α gates. The horizontal axis is the baseline signal energy; an engineer would assess end-user requirements and pick a signal energy for an uncorrected circuit on the basis of $p_{\text{error}} = \exp(-E_{\text{signal}} / kT)$. For example, this signal energy might be 60-100 kT for a supercomputer but only 40 kT for a consumer device.

The top magenta (E) and dark blue (E') curves represent system energy without and with error correction respectively. The magenta curve for E is represents the exponential Landauer-Shannon limit, although accounting for error propagation. Error correction allows signal energy to be reduced at the expense of overhead for the error correction logic. The dark blue curve (E') shows total system energy at constant error rate. The blue curve is lower on the right of the graph, asymptotically approaching about 2:1 reduction in energy for the 7-input gate case. The error correction stops working below 15 kT or so, meaning the error-corrected circuit consumes more energy for the same output error rate.

The limiting factor at low signal energy for error correction is illustrated by the yellow (E_x'+E_c') curve representing energy consumed by the errors and error correction circuitry. The blue-green (E_x) curve represents the energy consumed by the errors themselves in the uncorrected case. The number of errors rises exponentially as signal energy drops. Furthermore, errors propagate through the circuit with fanout such that each error produces a large number of downstream errors in the circuit. Both the yellow and green curves include an exponential rise when moving to the left.

We conclude that there is an opportunity to exceed the purported Landauer-Shannon limit. However, the upside potential depends on many variables. Even ideal millivolt switches that have no leakage will consume power due to the energy of creating and propagating thermal errors. This effect becomes dominant for logic nets operating below 20 kT in this example but varies based on fanout and circuit depth.

IV. Adiabatic Charge-Recycling Memory

Aside from energy efficiency improvements resulting from monolithic integration of memory and logic, adiabatic charge-recycling has been explored to further increase energy efficiency for matrix-vector multiplication in massively parallel connectionist neural computation [16]. It saves substantial energy by conserving charge through capacitive coupling, rather than destructive charge transfer.

Here, we investigate the energy efficiency of the same adiabatic charge-recycling circuit when used in a memory. Since the proposed memory uses the same adiabatic circuit that was fabricated, tested, and reported as a neural network array processor [16], will report on its performance in the context of

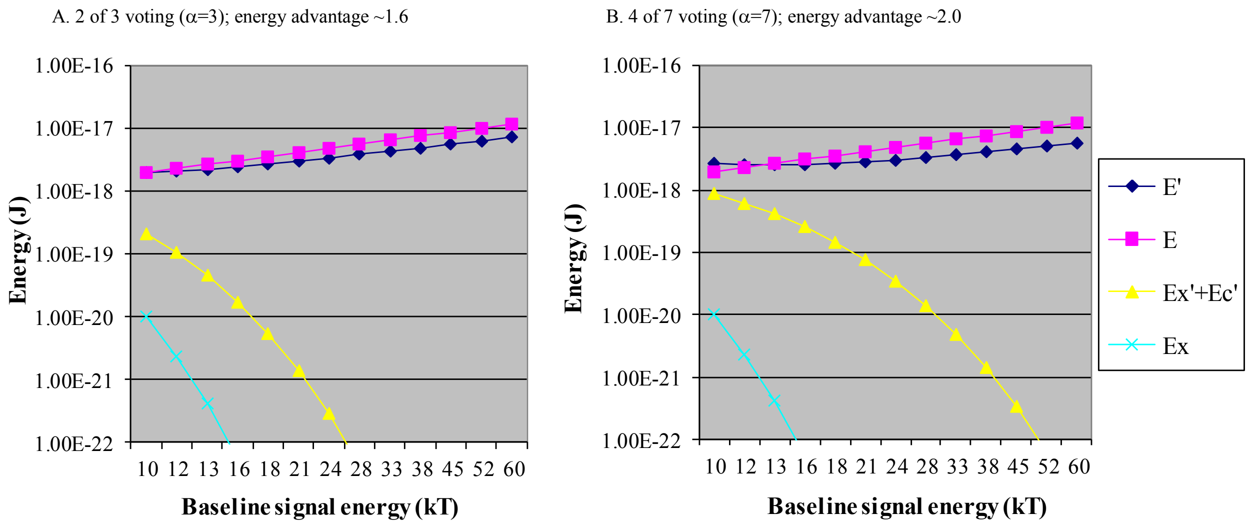


Fig. 2. Modeling of temporal error correction, α=3 and 7

a memory by reinterpreting past experimental results.

Memories dissipate energy in the addressing logic, capacitance between row and column conductors and ground, and in storage devices. The approach illustrated in Fig. 3 reduces all these energy losses by recovering energy recirculating between electrostatic and inductive forms in an LC tank circuit at its resonant oscillation frequency.

Fig. 3A illustrates the principle using a small memory array with columns in green, rows in orange, and charge-injection device (CID) storage cells in grey rectangles. Each CID cell is abstracted as an open circuit for a 0 and full-charge capacitive coupling for a 1. The current flow path for a read cycle is shown in blue. Energy stored in the inductor is connected to one row at a time through the addressing switch. The remainder of the circuit is an arrangement of one or four capacitors depending on the state of the storage cell. The blue-indicated elements form a tank circuit, which will oscillate at its resonant frequency. The principle of adiabatic resonant energy recovery is as follows: As long as the timing of the opening and closing of the addressing switch is properly synchronized with the oscillation, the resonant energy in the tank is conserved as charge is circulated, except for minor energy losses due to parasitic resistances.

Output data is sensed from the columns by a charge-sensitive amplifier. The amplifier detects a voltage change when the CID is in the fully charged capacitive form, but there is no voltage in the open circuit form.

The CID cell is realized by two charge-coupled MOSFETs in series (Fig. 3B, upper). The two MOSFET gates are connected to the row and column conductors. Since the two MOSFETs are isolated from their surroundings except for purely capacitive output coupling, charge is mostly conserved. Charge in the yellow isolation area in Fig. 3B can leak out through the gate oxide or reverse biased diodes that may exist in the substrate area, but this leakage is relatively slow and can be addressed with DRAM-like refresh (at Hz to kHz rates).

The CID-equivalent circuit (Fig. 3B, lower) changes from an open circuit to a nonlinear capacitor based on charge in the isolation region. With no stored charge (Q=0) both transistors are in a non-conductive state, and the equivalent circuit is an open circuit. On the other hand, a fully charged cell ($Q=Q_{max}$) effectively couples all its charge between the gates.

A memory chip would have single inductor for many banks with one row driven in each, in which case the tank's capacitor comprises total capacitance of the fully charged CID cells in all the selected rows at a given instant. The total capacitance and therefore the resonant frequency depends on the stored data. It is critical that the LC tank is maintained at its resonance peak for most efficient energy recovery.

The adiabatic circuit described above was fabricated and measured [16] as a 256×256 array multiplier, which would have similar circuit characteristics to a memory chip with 256 banks. Fig. 4 shows the impact of the energy recycling. The upper red lines are without energy recycling, which makes them comparable to production memories. With recycling turned on and tuned, energy consumption shown by the blue curves dropped by up to 85×, peaking when half the selected CID cells were storing 0 and the other half storing 1.

The adiabatic resonant energy recovery principle applies specifically to charge-based memory storage with capacitive readout, as described above. It readily extends to other memory types where charge can be inherently conserved—however, emerging resistive memory technologies such as memristor (ReRAM), PCRAM, and MRAM crossbars are inherently lossy and do not permit adiabatic energy recovery.

V. Discussion

Both error correction methods above rely on cooperation across multiple technology levels. Moore's Law presupposed improvement only at the individual device level, assuming it would be a "rising tide that lifts all ships" without redesign of the ships. However, the error correction methods above recognize that the desirable reduction in size and energy of devices results in an undesirable increase in error rate. These errors cannot be corrected at the device level with increasing energy levels, but the aggregate result of these errors can be corrected at higher levels of a computer's technology stack.

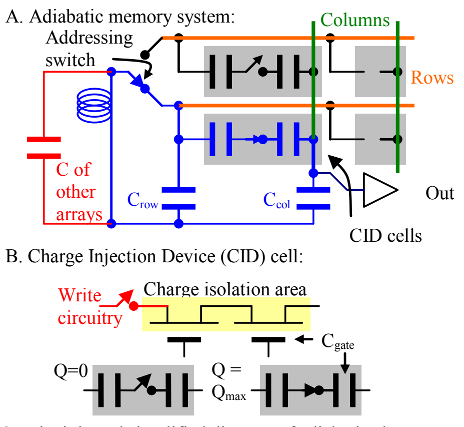


Fig. 3. Principle and simplified diagram of adiabatic charge-recovery

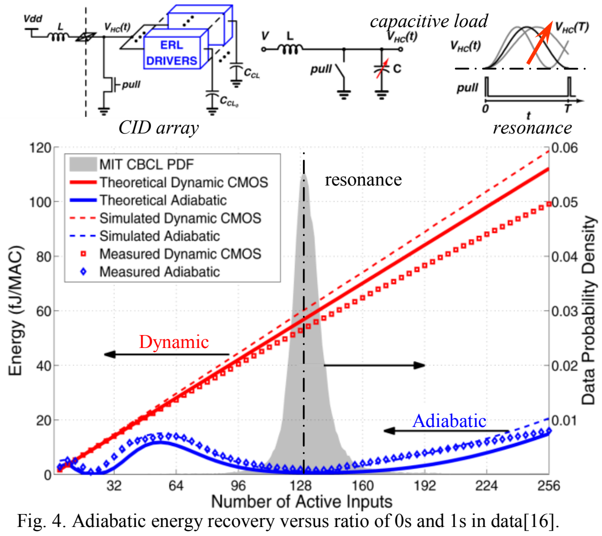


Fig. 4. Adiabatic energy recovery versus ratio of 0s and 1s in data[16].

The RRNS example uses mathematics to allow occasional errors to be corrected. Temporal error correction uses more than one gate's worth of energy to fix an error, but it can be applied to many gates at a time—leading to net advantage.

We can construct error correctoin schemes that beat the Landauer-Shannon limit, but we do not know the limits of the approach. Some applications require extremely reliable computing at the user level, such as Exascale supercomputers that may not make a single mistake over a multi-year lifetime. The simple analyses in this paper suggest 2× advantage for supercomputers. On the other hand, users watching video on a smartphone are likely to tolerate a bad pixel once in a while so the reliability requirements are less. The energy reduction due to error correction will be less in this case.

The RRNS example in section II was based on number system devised in 1965 for significantly different purpose. A separate paper in this conference details work in devising number systems and architectures that may perform better [13]. However, we do not know the potential of the approach.

The temporal error correction in section III was susceptible to direct analysis. It had two energy levels: E_{signal}' for the logic and E_{signal} for the correction circuitry. In the opinion of the authors, temporal error correction to reduce energy is unlikely to become a specific circuit or device. It seems more likely that a computerized design tool might be able to optimize a logic layout for low energy by applying algorithms to each gate.

Adiabatic memory shows significant promise in an unusual area. The memory in today's computers does not produce much heat, but there is a lot of interest in computer applications that use a lot of data. In conjunction with emerging 3D manufacture of logic or memory, it is possible that today's low power single-layer memories will evolve to hundred-layer modules that dissipate a hundred times as much power. After a 100× rise, memory would no longer be low power. This would create a demand for energy-efficient memories such as described.

VI. Conclusions

By simple arithmetic, an Exascale supercomputer needs an uncorrected gate-level reliability equivalent to an E_{signal} of around 60 kT to avoid silent errors over its lifetime. In this range, the error correction described in this paper could reduce overall energy by around 2×, for a effective energy of 30 kT. Using [17] as reference, logic is heading towards an energy of, say, 10,000 kT per operation. Subject to evolution of transistors into millivolt switches (which is not assured), the remaining improvement would be 10,000 / 30 ≈ 300×.

Reversible logic seems to be emerging as a practical option for continued scaling, including both reversible processors [12] and the discussion of reversible memory in this article. It is likely that the energy efficiency of memories will improve as Moore's Law progresses, yet the CV^2 energy in the row/column lines will limit energy efficiency. The adiabatic approach discussed in this paper demonstrated an 85× boost in a laboratory demonstration that could be further improved. The energy reduction from adiabatic operation would combine multiplicatively with the improvement due to Moore's Law.

More study of the limits of current technology would seem indicated. The semiconductor industry spends billions of dollars on new fab lines to get each additional 2× energy efficiency. Theory work on error correction appears from this paper seem capable of getting the same result at reduced cost.

Continued research on millivolt switches is indicated. This paper shows the devices would make even more of a contribution than currently expected if accompanied by error correction.

Memory is moving to 3D now, which has obvious benefits for both energy efficiency and applications that may need to use a lot of memory. However, device physics research would be needed for memory cells that avoid dissipating large amounts of power during reads and write—such as the open circuit/capacitor CID cell discussed in the paper.

References

[1] G. Moore, "Cramming more components onto integrated circuits, Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff."

[2] R. Landauer, "Irreversibility and heat generation in the computing process," IBM journal of research and development 5.3 (1961): 183-191.

[3] E. DeBenedictis, M. Frank, N. Ganesh, N. G. Anderson, "A path toward ultra-low-energy computing," International Conference on Rebooting Computing, 2016.

[4] M. Neyman, "The negentropy principle in information-processing systems," Telecommunications and Radio Engineering 21 (1966): 68.

[5] J. Bellamy, Digital telephony (Wiley Series in Telecommunications and Signal Processing), Wiley-Interscience, 2000.

[6] T. Theis and P. Solomon, 'In quest of the "next switch": prospects for greatly reduced power dissipation in a successor to the silicon field-effect transistor,' Proceedings of the IEEE 98.12 (2010): 2005-2014.

[7] R. Keyes and R. Landauer, "Minimal energy dissipation in logic," IBM Journal of Research and Development 14.2 (1970): 152.

[8] Robert Lyons and Wouter Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," IBM Journal of Research and Development 6.2 (1962): 200-209.

[9] Kuang-Hua Huang and Jacob A. Abraham, "Algorithm-based fault tolerance for matrix operations," IEEE transactions on computers 100.6 (1984): 518-528.

[10] C. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development* 17.6 (1973): 525-532.

[11] M.P. Frank, "Reversibility for efficient computing," Ph. D. thesis, Massachusetts Institute of Technology, 1999.

[12] César O. Campos-Aguillón, et. al., "A Mini-MIPS microprocessor for adiabatic computing," International Conference on Rebooting Computing, 2016.

[13] B. Deng, et. al., "Computationally-Redundant Energy-Efficient Processing for Y'all (CREEPY)," International Conference on Rebooting Computing, 2016.

[14] R. Watson and C. Hastings, "Self-checked computation using residue arithmetic," Proceedings of the IEEE 54.12 (1966): 1920-1931.

[15] E. DeBenedictis and H. Zima, "Millivolt switches will support better energy-reliability tradeoffs," Energy Efficient Electronic Systems (E3S), 2015 Fourth Berkeley Symposium on. IEEE, 2015.

[16] R. Karakiewicz, R. Genov, and G. Cauwenberghs, "1.1 TMACS/mW fine-grained stochastic resonant charge-recycling array processor," Sensors Journal, IEEE 12.4 (2012): 785-792.

[17] D. Frank, "Reversible adiabatic classical computation—an overview," 2nd IEEE Rebooting Computing Summit, 2014, slide 23; http://rebootingcomputing.ieee.org/images/files/pdf/7-rcs2-adiabatic-reversible-5-14-14.pdf.

Computationally-Redundant Energy-Efficient Processing for Y'all (CREEPY)

Bobin Deng*, Sriseshan Srikanth*, Eric R. Hein†, Paul G. Rabbat† and Thomas M. Conte*†
*School of Computer Science
†School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
Email: {$bdeng, seshan, heine, prabbat3, conte$}@gatech.edu

Erik DeBenedictis‡ and Jeanine Cook‡
‡Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
Email: {$epdeben, jeacook$}@sandia.gov

***Abstract*—Dennard scaling has ended. Lowering the voltage supply (Vdd) to sub volt levels causes intermittent losses in signal integrity, rendering further scaling (down) no longer acceptable as a means to lower the power required by a processor core. However, if it were possible to recover the occasional losses due to lower Vdd in an efficient manner, one could effectively lower power. In other words, by deploying the right amount and kind of redundancy, we can strike a balance between overhead incurred in achieving reliability and savings realized by permitting lower Vdd. One promising approach is the Redundant Residue Number System (RRNS) representation. Unlike other error correcting codes, RRNS has the important property of being closed under addition, subtraction and multiplication. Thus enabling correction of errors caused due to both faulty storage and compute units. Furthermore, the incorporated approach uses a fraction of the overhead and is more efficient when compared to the conventional technique used for compute-reliability.**

In this article, we provide an overview of the architecture of a CREEPY core that leverages this property of RRNS and discuss associated algorithms such as error detection/correction, arithmetic overflow detection and signed number representation. Finally, we demonstrate the usability of such a computer by quantifying a performance-reliability trade-off and provide a lower bound measure of tolerable input signal energy at a gate, while still maintaining reliability.

I. Introduction

Dennard scaling [6] has been one of the main phenomena driving efficiency improvements of computers through several decades. The main idea of this law is that transistors consume the same amount of power per unit area as they scale down in size. However, leakage current and threshold voltage limits have caused Dennard scaling to end [2]. This essentially negates any performance benefits that Moore's law may provide in the future; power considerations dictate that a higher transistor density results in either a lower clock rate or a reduction in active chip area.

In this paper, we propose a scalable architectural technique to effectively extend the performance benefits of Moore's law. We enable reducing the supply voltage beyond conservative thresholds by efficiently correcting intermittent computational errors that may arise from doing so. Energy benefits are observed as long as the overhead incurred in error correction is less than that saved by lowering V_{dd}. Furthermore, it may also be possible to lower energy by using *marginal*/post-CMOS devices, but such devices may sometimes be unreliable (due to tunneling effects, for instance [1]). The CREEPY approach of introducing error correcting hardware to lower energy is therefore deemed beneficial.

Figure 1 depicts a prior study conducted by one of the authors that estimates the potential of such a CREEPY computing paradigm in the future.

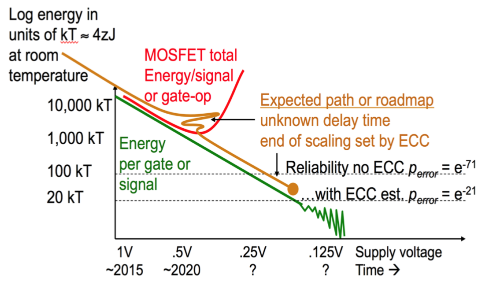


Fig. 1: Energy consequences of the fact that the error probability increases exponentially with decrease in signal energy. Here, the term ECC encapsulates any error correction mechanism in general.

We first provide some mathematical background that forms the basis of our proposed architecture.

II. Background

A. Triple Modular Redundancy (TMR)

The conventional approach to computational fault tolerance is TMR [9]; the idea is to replicate the hardware twice (for a sum total of three computations per computation) and then take a majority vote. With a model that assumes that at most one of these three computations can be in error at any given point in time, it follows that at least two of the computations are error-free; this can thus be used to detect and correct a single error, assuming an error-free voter.

While simple to understand and implement, this introduces an overhead of 200% in area and power, which leaves plenty of room for improvement. Any energy savings from lowering

TABLE I: A (4, 2)-RRNS example with the simplified base set (3, 5, 2, 7, 11, 13). Range is 210, with 11 and 13 being the redundant bases.

Decimal	mod 3	mod 5	mod 2	mod 7	mod 11	mod 13
13	1	3	1	6	2	0
14	2	4	0	0	3	1
13+14=27	(1+2)mod 3=0	2	1	6	5	1
	All columns (residues) function independently of one another.					
	An error in any one of these columns (residues) can be corrected by the remaining columns.					

V_{dd} and/or using post-CMOS devices would be eclipsed due to this overhead in correcting resultant errors.

B. Residue Number System (RNS)

The Residue Number System [7] has been used as an alternative to the binary number system chiefly to speed up computation, especially in signal processors [4], [5], [11]. This increased efficiency comes from the fact that a large integer can be represented using a set of smaller integers, with arithmetic operations permissible on the set in parallel (with the exception of division, comparison and binary bit manipulation). We present some of the properties of an RNS system without proof.

Let $B = \{m_i \in \mathbb{N}\ for\ i = 1, 2, 3, ..., n\}$ be a set of n co-prime natural numbers, which we shall refer to as bases or moduli. $M = \prod_{i=1}^{n} m_i$ defines the range of natural numbers that can be injectively represented by an RNS system that is defined by the set of bases B. Specifically, for x such that $x \in \mathbb{N},\ x < M$, then, $x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n})$, where $|x|_m = x\ mod\ m$. Each term in this n-tuple is referred to as a residue.

We also note that addition, subtraction and multiplication are closed under RNS. This is because of the following observation: given $x, y \in \mathbb{N},\ x, y < M$, we have $|x\ op\ y|_m = ||x|_m\ op\ |y|_m|_m$, where op is any add/subtract/multiply operation. Unsupported arithmetic operations (including division), thereby, would incur a performance and energy overhead; such operations should be carefully handled by the compiler, the specifics of which warrant further research and are beyond the scope of this paper.

C. Redundant Residue Number System (RRNS)

To augment RNS with fault tolerance, r redundant bases are introduced [8], [12], [13]. The set of moduli now contains n non-redundant and r redundant moduli: $B = \{m_i \in \mathbb{N}\ for\ i = 1, 2, 3, ..., n, n+1, ..., n+r\}$. The reason these extra bases are redundant is because any natural number smaller than M $(= \prod_{i=1}^{n} m_i)$ can still be represented uniquely by its n non-redundant residues. Intuitively, the r redundant residues form a sort of an *error code* because of the fact that all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N},\ x < M$, then, $x \equiv (|x|_{m_1}, |x|_{m_2}, |x|_{m_3}, ..., |x|_{m_n}, |x|_{m_{n+1}}, ..., |x|_{m_r})$ contains n non-redundant residues as well as r redundant residues.

Upon applying arithmetic transformations to an RRNS number, any error that occurs in one of the residues is contained within that residue and does not propagate to other residues. When required, such an error can be corrected with the help of the remaining residues. Specifically, an RRNS system with (n, r) = (4, 2), a single errant residue can be corrected, or, two errant residues can be detected. Table I provides a simple example, Section IV-D outlines necessary algorithms to do the single error correction. Research by Watson and Hastings [8], [12], [13] lays the foundation for the underlying theoretical framework that is used and extended in our work. We use (199, 233, 194, 239, 251, 509) as our (4, 2)-RRNS system, providing a range $M = 199 \times 233 \times 194 \times 239 \in (2^{31}, 2^{32})$. These RRNS moduli were chosen to fit in 8-bit or 9-bit registers.

It can be seen that a redundant residue number system achieves a higher efficiency due to enhanced bit-level parallelism while also providing resilience with only 50% overhead. As the granularity of an error is that of an entire residue, RRNS is capable of potentially correcting multi-bit errors as well. Moreover, RRNS could efficiently handle the error chaining problem; if we are summing up an array and a single add is in error, we can fix the final sum at the end with a single check.

We now design a computer using these properties.

III. CREEPY Overview

Given the potential of *marginal* devices, *i.e.,* post-CMOS/millivolt switches that are conducive to lowering voltages (reliability deprecates, but gracefully), CREEPY aims to achieve lower energy in high throughput exascale HPC systems by lowering the supply voltage while efficiently correcting the resulting errors.

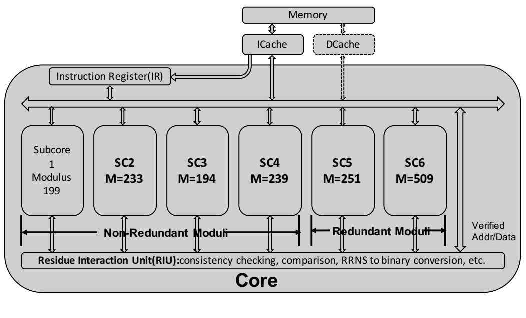


Fig. 2: The CREEPY Core with the reference RRNS system. The register file and data cache are distributed across subcores.

A CREEPY core consists of 6 *subcores*, an Instruction Register (IR) and a Residue Interaction Unit (RIU), as depicted in Figure 2.

Each subcore is fault-isolated from the others because it is designed to operate on a single residue of data. This can be thought of as analogous to a bit-slice processor. After posting a successful instruction fetch (the instruction cache stores instructions in binary, and is ECC protected), the checked instruction is dispatched onto the 6 subcores, which then proceed to operate on their corresponding slice of data. For example, adding two registers is done on a per residue basis; the register file is itself distributed across the 6 subcores. Similarly, the data cache is also distributed across the 6 subcores and stores RRNS protected data. The RIU is then responsible to perform any operations that involve more than a single residue, such as RRNS consistency checking, comparison and conversion to binary.

CREEPY employs standard ECC[10] to protect the main memory because of ECC's compactness and efficiency when it comes to protecting stored data. As such, both data and instructions are simply 32 bits each, not counting their ECC protection. However, the 32 bit representation of data is in RNS form (as opposed to binary). The memory controller checks ECC on a processor load and generates the two redundant residues before loading data into the last level cache. Similarly, it generates ECC upon a processor store (and the redundant residues are not stored into the main memory).

The focus of this paper is on the architecture of a computationally error tolerant core/processing element and *not* on ECC techniques for reliable storage.

IV. CREEPY CORE DESIGN

In this section, we present several aspects of a CREEPY core design.

A. Instruction Set Architecture(ISA)

In order to simplify instruction fetch and decode stages, all instructions are fixed to 32 bits wide. The ISA expects 32 registers (R0-R31), with R0 hard-wired to zero, R30 being the link register and R31 storing the default next PC ($R31 = PC + 4$). In our micro-architecture, each register is 49 bits long (*i.e.,* it contains the RRNS redundant residues as well) and is sliced on a per-modulus (sub-core) basis. The data cache is also implemented in a similar manner, as it stores data in an RRNS format. We discuss the formats of several important CREEPY instructions in the remaining part of this section.

1) R-Format (ADD/SUB/MUL)
These instructions assume that the destination operand as well as both source operands are registers.

Opcode	Src Reg1	Src Reg2	Dest Reg	Reserved
6b	5b	5b	5b	11b

2) I-Format (ADDI/SUBI/MULI)
For instructions that require compiler generated immediate literals, two new instructions (that always occur in succession without exception) are defined. Telescopic op-codes are employed to facilitate implementation of such *set* instructions. The fundamental need for the *set* instruction arises from the fact that literals are 49 bit RRNS values and would not otherwise simply fit within a 32 bit field (next to an immediate instruction, for example).
Set123 sets the the first 3 residues of the immediate value into the first 3 sub-core slices of the destination register and *Set456* sets the remaining 3 residues of the immediate value into the other three sub-core slices of the destination register.

Opcode	Dest	Reserved	Residue3	Residue2	Residue1
11[2b]	5b	0[1b]	8b	8b	8b

Opcode	Dest	Residue6	Residue5	Residue4
11[2b]	5b	9b	8b	8b

For an example, consider the immediate instruction *Addi R1, R2, 0x020202020202*. A CREEPY program would implement this instruction as follows:

a) Set123 R3, 020202
b) Set456 R3, 020202
c) Add R1, R3, R2

3) Branch

Opcode	Reg1	Reg2	Reg3	Link	Reserved
6b	5b	5b	5b	1b	10b

Recall that R0 = 0, R31 = PC + 4 and that R30 is the link register. A CREEPY branch follows one of the following semantics:

a) Reg1 = R0 and Reg3 = R0 and Link = 0: An unconditional branch that always jumps to the address in Reg2.
b) Link = 0: A conditional branch that jumps to the address in Reg2 (base) + Reg3 (offset) if Reg1 is 0. This is otherwise known as a *beqz* instruction.
c) Link = 1: A branch and link instruction to enable sub-routine calls and returns. The default next PC is stored into the link register and the program jumps to the address in Reg2.

4) Load/Store

Opcode	Reg1	Reg2	Reg3	Reserved
6b	5b	5b	5b	11b

Reg3 is the destination for a load and also is the source register for a store. The source/destination address for a load/store is given by Reg1 (base) + Reg2 (offset). Note that the memory address is hereby stored in an RRNS format.
Helper instructions such as *mov* etc. also exist, but are omitted from this description for brevity.

B. Error Model

First, we distinguish fault, error and failure as follows:
Fault: A single bit flips, but is not stuck-at, *i.e.,* only intermittent / transient faults are considered. Causes may range

from unreliable devices to low supply voltage to particle strikes to random noise and any combination therein.

Error: One or more faults in a single residue that show up during a consistency check.

Failure: The system has at least one error that it cannot detect, or has detected and cannot correct.

Faults may lead to errors which may lead to failures. We can guarantee the system is reliable if at most one error per core occurs between two consistency checks.

Redundancy in time, *i.e.*, check at cycle x, check again at cycle y, check again at cycle z, and vote, does not apply to this model as it is possible that the three checks suffer 3 independent 1 bit faults, rendering voting useless. The transient clause in the model rules out stuck-at faults. An implication of this is that we cannot achieve reliability by merely trading performance alone. Additional resources in terms of spatial redundancy are necessary, which is exactly what has been designed.

Different components of the core are protected via specialized means that target each component. The guiding principle is to design a system that uses the more efficient of RRNS/ECC based redundancy based on the range and nature of data being protected. Where both techniques are deemed insufficient to prevent the fault from metastasizing into an error, and eventually into a failure, the more conventional (and expensive) method: Triple Modular Redundancy (TMR), is employed. An alternative is to prevent the fault from occurring in the first place by using high V_{dd} (and/or circuit hardening). Choosing optimally between the latter expensive techniques is beyond the scope of this paper, but we assume that control signals' integrity is ensured using either TMR or intelligent state assignment, and that the RIU uses a high V_{dd} / hardened circuitry.

C. Signed Numbers Representation

In this section, we describe three competing ways of implicitly representing signed numbers, the first two of which were proposed by Waston [12], which we term Complement $M \times MR$ and Complement M representations, where, M is the product of all the non-redundant moduli ($M = m1 \times m2 \times m3 \times m4$) and MR is the product of all the redundant moduli ($MR = m5 \times m6$). To make up for the fact that Complement $M \times MR$ breaks the error correction algorithms and that Complement M is generally poor in performance, we propose a third approach, which we refer to as Excess-$\frac{M}{2}$ representation.

1) **Complement $M \times MR$ Signed Representation**
The $M \times MR$ complement signed representation is depicted by Figure 3. To provide a few examples, 0 is represented by 0, 1 is represented by 1, $\frac{M}{2} - 1$ is represented by $\frac{M}{2} - 1$, -1 is represented by $M \times MR - 1$ and $-\frac{M}{2}$ is represented by $M \times MR - \frac{M}{2}$. As can be seen, this is similar to signed binary representation. However, the known error correction algorithms break if numbers are represented in this manner.

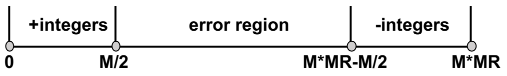


Fig. 3: Complement $M \times MR$ signed representation

2) **Complement M Signed Representation**
The complement M signed representation is depicted in Figure 4. This is similar to the $M \times MR$ method, except that the wrap-around occurs at M as opposed to $M \times MR$. This representation does not break error correction algorithms, provided that some correction factors (scaling and offset) are applied to the result of each arithmetic operation. However, further analysis indicates that calculating these correction factors [1] require knowledge of the signs of the operands, which is not explicitly known and sign determination in RRNS is a time-consuming process.

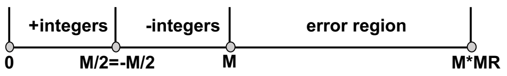


Fig. 4: Complement M signed representation

3) **Excess-$\frac{M}{2}$ Signed Representation**
The Excess-$\frac{M}{2}$ signed representation is shown in Figure 5. The excess notation, sometimes known as offset notation, merely shifts each number by $\frac{M}{2}$. To further elaborate, 0 is represented by $\frac{M}{2}$, 1 is represented by $\frac{M}{2} + 1$ and -1 is represented by $\frac{M}{2} - 1$. Similar to the M Complement representation, the results of arithmetic operations must be offset by a correction factor before they can be corrected. However, these correction factors turn out to be independent of the sign of the operands. So this method has great potential to improve the performance.
From the simulation results in V-D, we choose Excess-$\frac{M}{2}$ to be the defacto signed representation scheme for CREEPY.

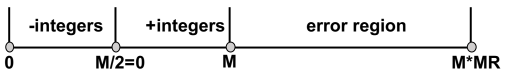


Fig. 5: Excess-$\frac{M}{2}$ signed representation

D. Residue Interaction Unit (RIU) Algorithms

The algorithm for single error correction was originally given by Watson [12]. However, RNS renders arithmetic overflow detection to be a non-trivial exercise. Furthermore, published work lacks sufficient details on the workings of overflow detection. In addition to providing a high level overview of the error correction algorithm, this section also presents algorithms for overflow detection. Furthermore, since the proposed algorithms augment the consistency checking algorithm itself, no extra hardware is warranted beyond that required by the error check.

[1]Correction factors are necessary to transform the results of an arithmetic operation before they can be checked for consistency. Details have been omitted due to space constraints.

1) Single Error Detection and Correction Algorithm: The single error detection and correction algorithm proposed by Watson [12] is based on an error correction table. The working steps of this algorithm for a system with 4 non-redundant moduli (m_1, m_2, m_3, m_4) and 2 redundant moduli (m_5, m_6), for any given integer X ($X < M = m_1 m_2 m_3 m_4$) is as follows: (a) Use a base-extension algorithm to compute $|X'|_{m_5}$ and $|X'|_{m_6}$, where $|X|_m$=X mod m. (b) For $i = 5, 6$: compute $\Delta m_i = |X'|_{m_i} - |X|_{m_i}$. (c) A non-zero difference indicates the presence of an error. This pair of differences indexes into an entry of a pre-computed (fixed) error correction table, which contains the index of the residue that is in error and a correction offset that needs to be added to that residue to correct said error. In CREEPY, the error checking may be delayed. In other words, error may be stored in a register and fixed later.

TABLE II: Error Correction table of RRNS System with Moduli (3,5,2,7,11,13)

Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ
1 , 10	4 6	4 , 11	4 5	7 , 7	4 3
2 , 10	2 3	5 , 8	4 4	7 , 8	1 2
2 , 12	4 6	5 , 9	2 1	8 , 1	2 2
3 , 3	1 1	5 , 12	3 1	8 , 4	4 2
3 , 9	4 5	6 , 1	3 1	8 , 10	1 2
3 , 12	2 3	6 , 4	2 4	9 , 1	4 1
4 , 5	1 1	6 , 5	4 3	9 , 3	2 2
4 , 6	4 4	7 , 2	4 2	10 , 3	4 1
4 , 7	2 1	7 , 6	4 2		

For ease of presentation, we present such an error correction table for a smaller (toy) set of RRNS base moduli in Table II. The total entries in such a table is at most $2\sum_{i=1}^{4}(m_i - 1)$. For the reminder of this section, these set of bases are used for explanatory purposes.

2) Unsigned Number Overflow Detection: In the absence of any error or overflow, adding 2 unsigned RRNS numbers results in (Δm_5,Δm_6) = (0,0). In the absence of error, we observe that any overflow manifests itself as a fixed index into the error correction table, with the entry not corresponding to any error. Table III provides some examples of this observation. While computations of the deltas are most efficient by using a base-extension algorithm, we use the Chinese Remainder Theorem (CRT) or the Mixed-Radix Conversion (MRC) method here to first convert the RRNS number to binary, before computing deltas. This is solely for explanatory purposes; binary conversion is not actually necessary to detect overflow.

Iterating through all possible combinations of numbers and operations, we observe that the value pair of (Δm_5, Δm_6) is fixed. Moreover, (Δm_5, Δm_6) = (10,11) is not a legitimate address of the error correction table (Table II), thus enabling a distinction between an error and an overflow. This approach, however, does not apply to multiplication.

3) Signed Number Overflow Detection: Recall from Section IV-C that CREEPY uses the Excess-$\frac{M}{2}$ signed representation. We discuss the two sources of overflow independently:

1) *Add two positive numbers.* Table IV provides a few examples illustrating the algorithm. The $1 + 104$ in the first column is represented in decimal. After Excess-$\frac{M}{2}$ mapping, the computing equation is transformed to $106 + 209$ since $\frac{M}{2} = 105$ for the toy set of moduli. Therefore, the X RRNS value is the the RRNS of 106 and Y RRNS value is the the RRNS of 209. We observe that the pair (Δm_5,Δm_6) remains at a fixed value (10,11).
2) *Add two negative numbers.* Similarly, examples for adding two negative numbers are shown in Table V. In this case, we observe that the pair (Δm_5,Δm_6) is fixed to (1,2).

Note that neither (10, 11) nor (1, 2) are legitimate addresses in Table II, thereby enabling a distinction between an error and an overflow. However, while this method works for both addition and subtraction, it does not hold for detection of multiplication overflow as the delta-pair is not constant and sometimes indexes into a legal error correction table entry.

Figure 6 shows the overview of the whole algorithm.

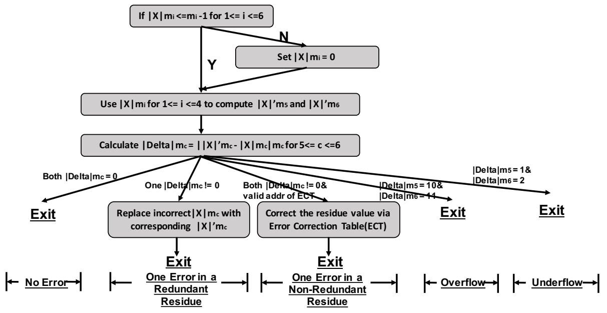

If |X|mi <=mi -1 for 1<= i <=6
N
Y
Set |X|mi = 0
Use |X|mi for 1<= i <=4 to compute |X|'m5 and |X|'m6
Calculate |Delta|mc = ||X|'mc - |X|mc|mc for 5<= c <=6
Both |Delta|mc = 0
One |Delta|mc != 0
Both |Delta|mc != 0& valid addr of ECT
|Delta|m5 = 10& |Delta|m6 = 11
|Delta|m5 = 1& |Delta|m6 = 2
Exit
Replace incorrect |X|mc with corresponding |X|'mc
Correct the residue value via Error Correction Table(ECT)
Exit
Exit
Exit
Exit
No Error
One Error in a Redundant Residue
One Error in a Non-Redundant Residue
Overflow
Underflow

Fig. 6: Single error detection and correction algorithm with overflow/underflow detection

We observe that the described algorithm works in a similar manner even with our original set of bases, (199,233,194,239,251,509). An overflow results in a delta-pair of (77, 289), whereas an underflow results in (174, 220). Both these pairs do not index into legitimate entries of the error correction table for these set of bases (*cf.* Appendix E, Watson [12]).

V. Simulation

To measure the performance *vs* reliability trade-off of CREEPY, we augment a stochastic fault injection mechanism into a cycle-accurate, in-order, trace-based timing simulator. We abstract the notion of using marginal devices and/or near threshold voltage into E_{signal} and P_e. E_{signal}, provided as an input to the simulation, is a measure of the signal energy at the input of a gate; P_e is the probability of a fault occurring at the output of a gate in any given cycle. The relationship of E_{signal} and P_e can be defined by the following relation: $P_e = \exp(\frac{-E_{signal}}{kT})$.

We first introduce a series of error events and their probabilities.

TABLE III: Unsigned Number Overflow Examples in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	CRT/MRC	$\|X'\|m_5$,$\|X'\|m_6$	Δm_5, Δm_6
2+209	(2,2,0,2,2,2)	(2,4,1,6,0,1)	(1,1,1,1,2,3)	$(1,1,1,1) \Leftrightarrow 1$	$\|1\|_{11}$=1,$\|1\|_{13}$=1	10 11
3+209	(0,3,1,3,3,3)	(2,4,1,6,0,1)	(2,2,0,2,3,4)	$(2,2,0,2) \Leftrightarrow 2$	$\|2\|_{11}$=2,$\|2\|_{13}$=2	10 11
...	...	...	...	...	...	10 11
209+209	(2,4,1,6,0,1)	(2,4,1,6,0,1)	(1,3,0,5,0,2)	$(1,3,0,5) \Leftrightarrow 208$	$\|208\|_{11}$=10,$\|208\|_{13}$=0	10 11

TABLE IV: Excess-$\frac{M}{2}$ Overflow Examples for addition of two positive numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\|X'\|m_5$,$\|X'\|m_6$	Δm_5,Δm_6
1+104	(1,1,0,1,7,2)	(2,4,1,6,0,1)	(0,0,1,0,7,3)	(0,0,0,0,1,2)	$(0,0,0,0) \Leftrightarrow 0$	$\|0\|_{11}$=0,$\|0\|_{13}$=0	10 11
2+104	(2,2,1,2,8,3)	(2,4,1,6,0,1)	(1,1,0,1,8,4)	(1,1,1,1,2,3)	$(1,1,1,1) \Leftrightarrow 1$	$\|1\|_{11}$=1,$\|1\|_{13}$=1	10 11
...	...	...	...	...	...	...	10 11

TABLE V: Excess-$\frac{M}{2}$ Overflow Examples for addition of two negative numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\|X'\|m_5$,$\|X'\|m_6$	Δm_5,Δm_6
-1-105	(2,4,0,6,5,0)	(0,0,0,0,0,0)	(2,4,0,6,5,0)	(2,4,1,6,10,12)	$(2,4,1,6) \Leftrightarrow 209$	$\|209\|_{11}$=0,$\|209\|_{13}$=1	1 2
-3-104	(0,2,0,4,3,11)	(1,1,1,1,1,1)	(1,3,1,5,4,12)	(1,3,0,5,9,11)	$(1,3,0,5) \Leftrightarrow 208$	$\|208\|_{11}$=10,$\|208\|_{13}$=0	1 2
...	...	...	...	...	...	...	1 2

P_e: Probability of a fault occurring at the output of a gate in any given cycle, as already defined.

P_{add}: Probability of at least a single error in an adder (each sub-core has an adder). If there are N_{add} gates in an adder, the probability of each of these gates being free of error is $(1 - P_e)^{N_{add}}$. Therefore, P_{add}=1-$(1 - P_e)^{N_{add}}$. Similarly, P_{mul} is calculated. For multi-cycle operations, this definition holds as long as the output state of each gate is used exactly once for the operation, which is true for all of our operators.

P_{R_i}: Probability of at least 1 fault being present in a slice (sub-core) of register R_i between cycles t_1 and t_2, where, t_2 is the time at which the RRNS consistency of R_i is being checked and t_1 is the time at which the RRNS consistency of R_i was last established. As t_1 depends upon f_n (the check frequency, which we discuss later) and the dynamic instruction trace, we explicitly maintain a mapping of $LastCheckedCycle[R_i]$ in our simulator. Assuming an SRAM implementation of 8-bit wide R_i, the number of transistors is $8 \times 6 = 48$. The probability of R_i being error free for the entire duration of (t_1, t_2) is $(1 - P')^{48(t_2 - t_1)}$, where P' is the probability of an error occurring in the state of an SRAM transistor. Due to the nature of an SRAM device, any fault occurring in one of its transistors gets latched, resulting in a higher probability of an error (when compared with glitches in logic transistors getting masked if the glitch does not occur close to the clock edge). As such, we assume $P' = 100P\_e$. Therefore, $P_{R_i} = 1 - (1 - 100P_e)^{48(t_2 - t_1)}$.

P_{loadX}: Probability of at least 1 fault being present in the loaded value of address X. This is clearly analogous to P_{R_i}, except that we maintain a mapping of $LastStoredCycle[X]$ to determine the last time a consistent state at address X was ensured. However, P_{loadX} also encapsulates the probability of an error in the implicit computation of the address X itself (from its base and offset) during the execution of the load.

P_{SC}: Probability of at least 1 fault occurring in a sub-core from the last time it was checked. To illustrate, say an RRNS check was placed after the instruction $ADD\ R_3, R_2, R_1$. Then, $P_{SC} = 1 - (1 - P_{add})(1 - P_{R_2})(1 - P_{R_1})$.

P_C: Probability of exactly 1 error occurring in a CREEPY core. This translates to exactly 1 sub-core being in error (where the sub-core error itself may be of multi-bit form; RRNS can tolerate multi-bit flips within a single residue). Therefore, $P_C = C_6^1 \times P_{SC}(1 - P_{SC})^5$, where the combinatorial choose operator C_n^r enumerates the number of ways in which r items can be chosen from n distinct items.

P_C^0: Probability of no error in a CREEPY core from the last time it was checked. $P_C^0 = 6C_0 \times (1 - P_{SC})^6$.

P_C^{fail}: Probability of a CREEPY core failing. The current version of the CREEPY micro-architecture is unable to correct more than 1 error occurring in the core and defers recovery to a software checkpoint. As such, we deem $\geq$ 2 errors in the core as amounting to a failure. Therefore, $P_C^{fail} = \sum_{2 \leq r \leq 6} C_6^r \times P_{SC}^r(1 - P_{SC})^{6-r} = 1 - P_C^0 - P_C$.

We statically compile integer benchmarks from the SPEC 2006 suite to generate a dynamic instruction trace compatible with the CREEPY ISA. The estimated gate count number for each of the five 8-bit subcores (not including the D Cache) is 2036, and, 2353 for the 9-bit one. Upon feeding this trace to the simulator, after every n^{th} instruction, as governed by f_n, the check frequency, an RRNS check is simulated. Based on P_C and P_C^0, it is stochasticaly determined if an RRNS correction must also be simulated. In accordance with the algorithms described in Section IV-D), we designate 8 cycles for the RRNS check and an additional 1 cycle should an error be corrected. At the end of each RRNS check, $P_{C,i}^{fail}$ is used to determine if a failure is likely to occur at that cycle t_i. As such, we use a typically used reliability metric, Mean Time To Failure (MTTF), which can be defined as follows: $MTTF = \frac{Total\ Cycles}{CPU\ Frequency\ \times\ \sum_i P_{C,i}^{fail}}$.

In addition to varying E_{signal}, we explore following optimizations that are expected to improve reliability and performance:

a) : Vary check frequency f_n, where f_n denotes that every n^{th} instruction is checked for an RRNS error. Intuitively, checking very frequently (ex. f_1; checking every instruction) favors higher reliability, whereas checking very infrequently (ex. f_∞, or equivalently, f_0) favors higher performance.

b) : For $n > 1$, perform a pipeline check on up to n destination registers / memory locations. This check strategy, known as $pipe_n$ is similar to f_n in that the consistency check action only happens after the n^{th} instruction, but with the added action that checks all the output destinations for the past n instructions in a pipeline fashion. Assuming that the original check takes 8 cycles, the check of $pipe_n$ should be 8+(n-1) cycles.

c) : For $n > 1$, in addition to performing a check (either f_n or $pipe_n$) every n^{th} instruction, also perform a check at every store instruction to enhance memory (cache) reliability. For brevity, we turn on this optimization for all the results presented in this section.

A. *Performance vs Consistency Check Frequencies*

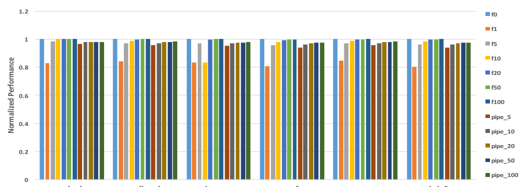


Fig. 7: Performances VS Consistency Check Frequencies

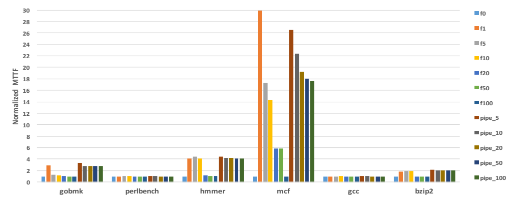


Fig. 8: MTTFs VS Consistency Check Frequencies

Figure 7 shows the relationship between system performance and consistency check frequency. We vary the check frequency from 0 to 100 in both non-pipeline and pipeline approaches. f_0 means that no extra consistency checks are performed whatsoever, and f_{100} performs an extra consistency check after every 100th instruction. Also recall that $pipe_5$ indicates that the results of each instruction are checked at the end of every 5^{th} instruction. The Y-axis is normalized against the baseline, which is f_0. The results show that the data trends of all the benchmarks are very similar. f_0 always gets the best performance because it incurs no consistency check overhead. Frequently checking for errors hurts performance, as evidenced by f_1 suffering from the worst performance degradation. Pipelined checks are very close to non-pipelined checks in performance on average. In other words, using the pipelined check approach enables a potentially more reliable system without sacrificing performance when compared to the non-pipelined check approach.

B. *MTTF vs Consistency Check Frequencies*

As explained earlier, Mean Time To Failure (MTTF) provides a measure of reliability. Similar to Section V-A, we normalize the MTTF against the baseline, f_0. Intuitively, a higher check frequency would result in a higher MTTF as the probability of an uncorrected error diminishes. However, it can be seen from Figure 8 that f_1 does not always provide the highest MTTF. This can be attributed to the fact that the consistency checks themselves are not instantaneous and this added delay leaves gates more vulnerable to faults.

For example, consider the f_1 and f_{10} scenarios below. All add instructions take 1 cycle and check instructions take 8 cycles. In instruction 0, R_5 will be checked in both cases. Then after this step, R_5 is first used in instruction 11. From the evaluation model we defined, the fault probability of R_5 depends on the time intervals between instruction 0 and 11 (time interval from last check). The time interval for the f_1 scenario in this example is $(1 + 8) \times 10 = 90$ cycles, but for f_{10}, this is only $1 \times 10 + 8 = 18$ cycles. Therefore, the fault probability of R_5 in instruction 11, is higher for the f_1 scenario than for f_{10}. Therefore, a low frequency pipeline checking (such as $pipe_5$ or $pipe_{10}$) achieves a good balance between performance and reliability.

```
$f_1$:
(0) add $R_5$, $R_2$, $R_3$
check $R_5$
(1) add $R_1$, $R_2$, $R_3$
 check $R_1$
(2) add $R_3$, $R_2$, $R_1$
check $R_3$
......//no check $R_5$
(10) add $R_8$, $R_2$, $R_1$
check $R_8$
(11) add $R_{11}$, $R_3$, $R_5$
```

```
$f_{10}$:
(0) add $R_5$, $R_2$, $R_3$
check $R_5$
(1) add $R_1$, $R_2$, $R_3$
(2) add $R_3$, $R_2$, $R_1$
......
(10) add $R_8$, $R_2$, $R_1$
check $R_8$
(11) add $R_{11}$, $R_3$, $R_5$
```

C. *MTTF vs Energy Input Per Gate*

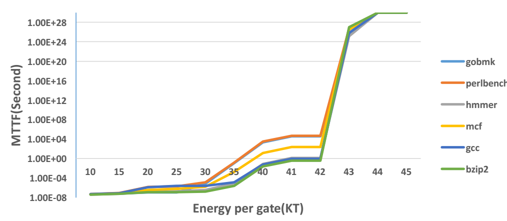


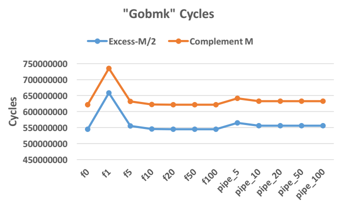
Fig. 10: MTTFs VS Energy input per gate

Figure 10 plots the simulated MTTF for various values of E_{signal}. For brevity, only the f_1 paradigm is depicted here. We notice that a value of E_{signal} greater than 44kT results in infinite MTTF, which is essentially a very large number, given the precision limits of the simulation. However, we also observe that the MTTFs drop very fast when the E_{signal} is between 42kT and 43kT.

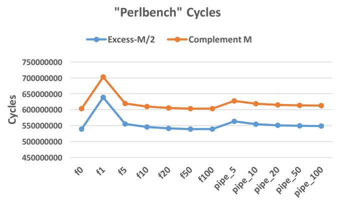
The key take away from Figures 7, 8 and 10 is that a lower E_{signal} can still lead to acceptable latency without degradation in output quality.

D. *Excess-$\frac{M}{2}$ vs Complement M signed representation*

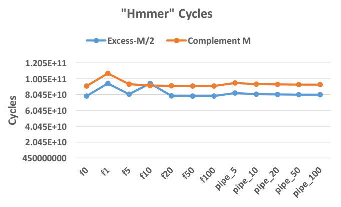
There is scope for further tuning the CREEPY core by employing alternate representations and algorithms. For instance,



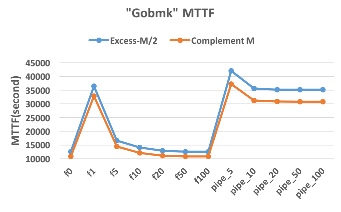
(a) Gobmk Performance



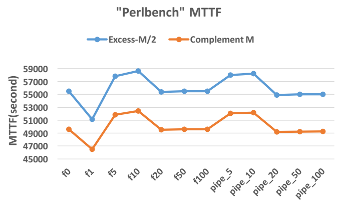
(b) Perlbench Performance



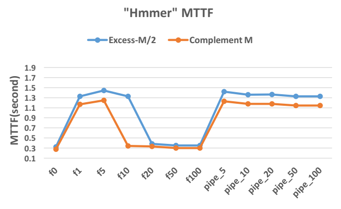
(c) Hmmer Performance



(d) Gobmk MTTF



(e) Perlbench MTTF



(f) Hmmer MTTF

Fig. 9: Performance and MTTF Comparisons for signed number representation methods: "Excess-$\frac{M}{2}$" and "Complement M"

there may be a set of bases that are conducive to very fast consistency checks. In this section, we compare two of the signed representations that were discussed in Section IV-C. The performance and reliability comparisons are shown in Figure 9. Recall that consistent arithmetic operations using the Complement M method require the need to determine the sign of operands, which is a time-consuming operation in RRNS. Therefore, the performance of Excess-$\frac{M}{2}$ is much better than Complement M on average. Even for MTTF, Excess-$\frac{M}{2}$ is better than Complement M. The reason is similar to that described in Section V-B (larger time intervals between consistency checks imply higher probability for system failure).

VI. Related work and Conclusion

While the concepts of RNS, RRNS, error correction, device limits and signal integrity by themselves are not new, we believe this is the first proposal that relates these mathematical and physical entities to push back the horizon of Moore's law for high-throughput exascale HPC systems. It must be noted that our approach does not require the algorithm to be designed in a fault tolerant manner, thereby expanding the scope of CREEPY to Turing completeness.

Prior work in the Digital Signal Processors (DSP) domain ([4], [5], [11]) has focused on RNS datapaths to take advantage of fine-grained data parallelism and energy efficient properties that RNS operations provide. By utilizing RRNS, CREEPY benefits from these, but improves upon generality and energy-efficiency. Chiang et al. [3] provide RNS algorithms for comparison and overflow detection, but assume all bases to be odd and do not consider error correction.

CREEPY, being an RRNS computer, draws its underlying mathematics heavily from the pioneering work of Watson and Hastings [8], [12], [13]. However, at the time, they probably did not deem it necessary to provide a detailed micro-architecture and ISA to support their algorithms. CREEPY extends and improvizes on their RRNS algorithms in addition to providing a detailed design. This paper describes and demonstrates the usability of a CREEPY computer that improves energy efficiency by adding computationally redundant hardware.

References

[1] D. E. Nikonov and I. A. Young, "Overview of Beyond-CMOS Devices and a Uniform Methodology for Their Benchmarking," in Proceedings of the IEEE, vol. 101, no. 12, pp. 2498-2533, Dec. 2013.

[2] McMenamin Adrian, *The end of Dennard scaling*, 2013.

[3] J.-S. Chiang and M. Lu, *Floating-point numbers in residue number systems*, Computers and Mathematics with Applications,22, no. 10, 127-140, 1991.

[4] Rooju Chokshi , Krzysztof S. Berezowski , Aviral Shrivastava , Stanislaw J. Piestrak, *Exploiting residue number system for power-efficient digital signal processing in embedded processors*, Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems, October 11-16, 2009, Grenoble, France

[5] E. Di Claudio, F. Piazza and G. Orlandi, *Fast Combinatorial RNS Processors for DSP Applications*, IEEE Trans. Computers, vol. 44, no. 5, pp. 624-633, May 1995.

[6] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, *Design of ion-implanted mosfets with very small physical dimensions*, IEEE Journal of Solid-State Circuits, 9, October 1974.

[7] H. L. Garner, *The Residue Number System*, IRE Transactions on Electronic Computers, pp. 140-147, June 1959.

[8] C. W. Hastings, *Automatic detection and correction of errors in digital computers using residue arithmetic*, in Proc. 1966 IEEE Region Six Annu. Conf, pp. 429-464.

[9] Lyons, Robert E., and Wouter Vanderkulk. "The use of triple-modular redundancy to improve computer reliability." IBM Journal of Research and Development 6.2 (1962): 200-209.

[10] MacWilliams F J, Sloane N J A, *The theory of error correcting codes[M]*, Elsevier, 1977.

[11] J. Ramirez, *RNS-enabled digital signal processor design*, Electron. Lett., vol. 38, no. 6, pp. 266-268, 2002

[12] R. W. Watson, *Error detection and correction and other residue interacting operations in a residue redundant number system*, Univ. California, Berkeley, 1965.

[13] R. W. Watson and C. W. Hastings, *Self-checked computation using residue arithmetic*, Proc. IEEE, vol. 54, pp. 1920-1931, Dec. 1966.

RRNS Overflow Detection and RRNS Comparison

In this document, we introduce an add/sub overflow detection algorithm during the consistency checking. This overflow detection method is suitable for both unsigned numbers (Section 2.1) and Excess –M/2 signed numbers (Section 2.2.3). Moreover, it doesn't need extra hardware overhead. Based on this overflow detection method, we also found a new residue numbers comparison algorithm (Section 3).

Watson briefly introduced a similar overflow detection method via consistency checking in page 109 of his thesis. But I don't think his method is correct. The detail discussions about these are in Section 2.2.1 and 2.2.2.

1. Single Error Detection and Correction Algorithm

In this section we simply introduce the SEC algorithm from section 2 of Watson's thesis, because the following overflow detection method may need some background information from it.

Input: The sequence of residues ($|X|_{m_1}$, $|X|_{m_2}$, … , $|X|_{m_{n+r}}$)
Output: A consistent sequence of residues ($|X|_{m_1}$, $|X|_{m_2}$, … , $|X|_{m_{n+r}}$)

Procedure

Step1: *Check to see that the condition ($|X|_{m_i} \leq m_i - 1$) is satisfied, $1 \leq i \leq n+r$.*
(a) If it is, then proceed to Step 2.
(b) If it is not, then set the residue which does not meet this condition to zero and proceed to Step 2.

Step2: *Use a base-extension algorithm to compute a new set of R residues from the information contained in the N-R residues.*

Step3: *Perform the calculations $|\Delta|_{mc} = ||X|'_{mc} - |X|_{mc}|_{mc}$ in a modulo m_c adder, where c = (n+1, …, n+r).*
*(a) If $|\Delta|_{mc} = 0$ for all values of c, then there is no error. **Exit**.*
*(b) If one and only one member of the sequence ($|\Delta|_{mc}$) is non-zero, then the R residue corresponding to the non-zero value of $|\Delta|_{mc}$ is incorrect. Replace this incorrect residue with the value of the corresponding residue $|X|'_{mc}$ computed by base extension. **Exit**.*
(c) If at least two members of the sequence ($|\Delta|_{mc}$) have a non-zero value, then the incorrect residue is in the N-R group of the residues. Proceed to Step 4.

Step4: *Use the r-tuple ($|\Delta|_{mc}$) as an address to an error correction table.*
(a) The output from the table will be the number of modulus of the N-R residue in error, i', and a value ε to be added to the incorrect residue in a modular adder to correct it. Proceed to Step 5.
(b) The output from the table will indicate that no such address is possible within the assumptions of this algorithm. Proceed to Step 6.

Step5: *Add the value ε obtained from the error correction table to the incorrect residue in a modulo m_i, adder, thus correcting it, where m_i, is the modulus of the incorrect N-R residue. **Exit**.*

***Step6:** Indicate that a more serious malfunction has occurred than is correctable with this algorithm. __**Exit**__.*

End Procedure

In Step1, it checks the condition ($|X|_{m_i} \leq m_i - 1$),where $1 \leq i \leq n+r$. In Creepy, the residues are encoded in binary. The moduli will not in general be powers of 2, and therefore it is possible that an error could occur in such a way as to make the integer represented by the binary code word greater than or equal to the modulus. If this condition occurs, it definitely indicates an error and can easily be detected with combinational logic. Because the SEC and DED algorithms are separated and we can only choose one of them. If choose DED, Step1 may early detect and handle errors. In SEC, this also definitely indicates an error. But no information is available at this point to tell what the magnitude of the correct residue should be. To obtain this extra necessary information, the rest of the algorithm must be followed. A value of a binary coded residue greater than or equal to the modulus does not make mathematical sense, so this SEC algorithm change them to zero once these errors are found.

In Step4, use the r-tuple ($|\Delta|_{mc}$) as an address to an error correction table. You can find the error correction table in Appendix E of Watson's thesis. It bases on moduli (199,233,194,239,251,509). Using value pair ($|\Delta|_{m5}$, $|\Delta|_{m6}$) as an address. There are M_R = m5* m6 = 251*509 = 127,759 possible combinations of the $|\Delta|_{mc}$. But Watson had proofed that the total number of correct table entries is only $2\sum_{i=1}^{n}(\mathrm{mi}-1)$. So the total entries of error correction table is (198+232+193+238)*2 = 1722. Thus only 1.4% of the possible pairs ($|\Delta|_{m5}$, $|\Delta|_{m6}$) are legitimate address to the error correction table. For simplify the overflow discussion in following section, I also generate the correct table for (3,5,2,7,11,13). Total entries of error correction table for system (3,5,2,7,11,13) is (2+4+1+6)*2 = 26. The examples in this document are based on moduli (3,5,2,7,11,13) unless otherwise specified.

TABLE1 ERROR CORRECT TABLE OF RRNS SYSTEM WITH MODULI (3,5,2,7,11,13)

$\lvert\Delta\rvert_{m5}$	$\lvert\Delta\rvert_{m6}$	i'	ε	$\lvert\Delta\rvert_{m5}$	$\lvert\Delta\rvert_{m6}$	i'	ε	$\lvert\Delta\rvert_{m5}$	$\lvert\Delta\rvert_{m6}$	i'	ε
1	10	4	6	4	11	4	5	7	7	4	3
2	10	2	3	5	8	4	4	7	8	1	2
2	12	4	6	5	9	2	1	8	1	2	2
3	3	1	1	5	12	3	1	8	4	4	2
3	9	4	5	6	1	3	1	8	10	1	2
3	12	2	3	6	4	2	4	9	1	4	1
4	5	1	1	6	5	4	3	9	3	2	2
4	6	4	4	7	2	4	2	10	3	4	1
4	7	2	1	7	6	2	4				

2. RRNS Overflow Detection by Consistency Checking

2.1 Unsigned Number Overflow Problem

The general way to detect overflow in RNS is via comparing the result of addition with one of the addends. If $X \geq 0$ and $Y < M$ then (X+Y) mod M causes overflow if and only if the result is less than X. Thus, we can conclude, that general overflow detection method is equivalent to the magnitude comparison.

One of the most efficient ways to detect overflow in RNS is via parity checking. It indicates whether an integer is even or odd. Suppose two integers (X, Y) have the same parity: Z = X + Y. An overflow occurs if Z is odd. Contrary, if (X, Y) have different parity, then an overflow occurs if Z is even. The parity checking technique is one of the best and fastest suggested methods to detect the overflow in RNS. However, this technique can only be used with moduli sets that have just odd members, i.e. odd dynamic range, which is not suitable for many moduli sets that uses even as one of its moduli. Thus, it is obvious that overflow detection in RNS that has an even dynamic range is a very important issue.

If we add 2 unsigned RRNS number, no error and no overflow, both $|\Delta|_{m5}$ and $|\Delta|_{m6}$ will be zero. But what would happen if overflow occurs but no error found?

E.g. in the system with moduli (3,5,2,7,11,13):

1) *2+209 = (2,2,0,2,2,2) + (2,4,1,6,0,1) = (1,1,1,1,2,3)*

Now we use base-extension algorithm or Chinese Reminder Theory to compute $|X|'_{m5}$ and $|X|,_{m6}$ base on the first 4 N-R moduli.

(1,1,1,1) <=> 1; $|X|'_{m5} = |1|_{11} = 1$; $|X|'_{m6} = |1|_{13} = 1$;

$(|X|_{m5}, |X|_{m6}) = (2, 3) \neq (|X|'_{m5}, |X|'_{m6})$

$\boldsymbol{(|\Delta|_{m5}, |\Delta|_{m6}) = (10,11)}$

From above example, it seems that consistent checking may detect overflow because $(|X|_{m5}, |X|_{m6}) \neq (|X|'_{m5}, |X|'_{m6})$. But if we use this method to detect overflow, how to distinguish error and overflow? See more examples:

2) *3+209 = (0,3,1,3,3,3) + (2,4,1,6,0,1) = (2,2,0,2,3,4)*

(2,2,0,2) <=> 2 ; $|X|'_{m5} = |2|_{11} = 2$; $|X|'_{m6} = |2|_{13} = 2$;

$(|X|_{m5}, |X|_{m6}) = (3, 4) \neq (|X|'_{m5}, |X|'_{m6})$

$\boldsymbol{(|\Delta|_{m5}, |\Delta|_{m6}) = (10,11)}$

3)209+209 = *(2,4,1,6,0,1) + (2,4,1,6,0,1) = (1,3,0,5,0,2)*

(1,3,0,5) <=> 208 ; $|X|'_{m5} = |208|_{11} = 10$; $|X|'_{m6} = |208|_{13} = 0$;

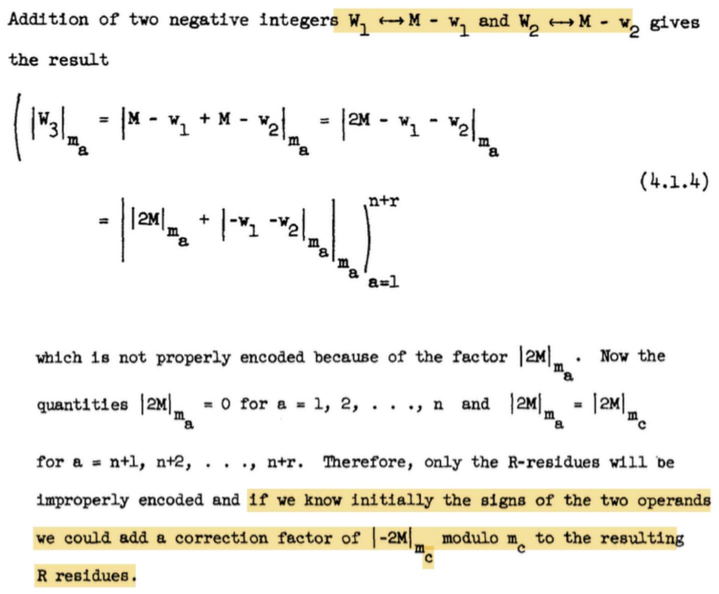
$(|X|_{m5}, |X|_{m6}) = (0, 2) \neq (|X|'_{m5}, |X|'_{m6})$

$\boldsymbol{(|\Delta|_{m5}, |\Delta|_{m6}) = (10,11)}$

I have tried all the possible combinations and find that $(|\Delta|_{m5}, |\Delta|_{m6})$ is a fixed value pair via a C program. $(|\Delta|_{m5}, |\Delta|_{m6}) = (10,11)$ is not a legitimate address of the error correction table(Table1), so we can distinguish error and overflow.

2.2 Signed Number Overflow Problem

2.2.1 Possible Error in page 102 of Watson's thesis (About correction factor values):



Addition of two negative integers $W_1 \longleftrightarrow M - w_1$ and $W_2 \longleftrightarrow M - w_2$ gives the result

$$\left(|W_3|_{m_a} = |M - w_1 + M - w_2|_{m_a} = |2M - w_1 - w_2|_{m_a} = \Big| |2M|_{m_a} + |-w_1 - w_2|_{m_a} \Big|_{m_a} \right)_{a=1}^{n+r} \qquad (4.1.4)$$

which is not properly encoded because of the factor $|2M|_{m_a}$. Now the quantities $|2M|_{m_a} = 0$ for a = 1, 2, . . ., n and $|2M|_{m_a} = |2M|_{m_c}$ for a = n+1, n+2, . . ., n+r. Therefore, only the R-residues will be improperly encoded and if we know initially the signs of the two operands we could add a correction factor of $|-2M|_{m_c}$ modulo m_c to the resulting R residues.

We can easily find that complement signed number representation method is used here because $W_1 \leftrightarrow M - w_1$ and $W_2 \leftrightarrow M - w_2$. w_1 and w_2 are positive numbers. But in equation 4.1.4, I think the correct final result should be $||M|_{m_a} + |M - w_1 - w_2|_{m_a}|_{m_a}$, not $||2M|_{m_a} + |-w_1 - w_2|_{m_a}|_{m_a}$. In other words, the correction factor for R-residues **should be $||-M|_{m_c}$, not $||-2M|_{m_c}$.**

E.g. (-3) = (-1)+(-2)

-3 $\leftrightarrow$ *210 -3 = 207* $\leftrightarrow$ *(0,2,1,4,9,12)*

-1 $\leftrightarrow$ *210 -1 = 209* $\leftrightarrow$ *(2,4,1,6,0,1)*

-2 $\leftrightarrow$ *210 -2 = 208* $\leftrightarrow$ *(1,3,0,5,10,0)*

(-1)+(-2) = (2,4,1,6,0,1)+ (1,3,0,5,10,0) = (3,7,1,11,10,1) = (0,2,1,4,10,1)

$|-M|_{11} = |-210|_{11} = 10$ $\quad |-M|_{13} = |-210|_{13} = 11$

$|-2M|_{11} = |-420|_{11} = 9$ $\quad |-M|_{13} = |-420|_{13} = 9$

if correction factor is $||-2M|_{m_c}$, result = *(0,2,1,4,10,1) + (0,0,0,0,9,9) = (0,2,1,4,8,10) ≠ (0,2,1,4,9,12)*

if correction factor is $||-M|_{m_c}$, result = *(0,2,1,4,10,1) + (0,0,0,0,10,11) = (0,2,1,4,9,12),* ***Correct!***

2.2.2 Complement (M/2 = -105) overflow detection

Watson briefly introduced a complement signed number overflow detection method via consistency checking in page 109 of his thesis. The method could be summarized like this:

1) *If addition of two positive operands or subtraction of a negative minuend from a positive subtrahend, the result will be in negative region if overflow occurs. So we can detect overflow by checking the signs.*

2) *If addition of two negative operands or the subtraction of a positive minuend from a negative subtrahend, the result will be in error region if overflow occurs. So we can use consistency checking to detect overflow.*

I don't think (2) is correct. Let's see an example for (2). Assuming that we add two negative numbers: (-105)+(-2) = -107.

-105(105) = (0,0,1,0,6,1)
-2(208) = (1,3,0,5,10,0)
-107 = (-105)+(-2) = (1,3,1,5,5,1) => add correct factor

if correction factor is **$|-M|_{m_c}$**, result = *(1,3,1,5,5,1) + (0,0,0,0,10,11) = (1,3,1,5,4,12)*
(1,3,1,5) <=> 103 ; $|X'|_{m5} = |103|_{11} = 4$; $|X'|_{m6} = |103|_{13} = 12$;
$(|\Delta|_{m5}, |\Delta|_{m6}) = (0,0)$, unable to detect overflow

if correction factor is **$|-2M|_{m_c}$**, result = *(1,3,1,5,5,1) + (0,0,0,0,9,9) = (1,3,1,5,3,10)*
(1,3,1,5) <=> 103 ; $|X'|_{m5} = |103|_{11} = 4$; $|X'|_{m6} = |103|_{13} = 12$;
$(|\Delta|_{m5}, |\Delta|_{m6}) = (1,2)$, detect overflow

But we have proofed in **Section 2.2.1** that **$|-2M|_{m_c}$** isn't the correct value. I think Watson used the wrong correction factor **$|-2M|_{m_c}$** to get his overflow detection conclusion (2). If the correction factor is **$|-M|_{m_c}$, we are unable to use consistency checking to detect overflow for complement.**

In complement representation, we still may use the expensive signs checking method to detect the overflow. Assume that we add two positive numbers: 104 + 2= 106. The overflow should happen because the range of sign number is [-105, 104]. 106 is mapped to -104 in complement method. In consistency checking, $(|\Delta|_{m5}, |\Delta|_{m6})$ will be (0,0) and unable to detect overflow. The 106 locates in negative region, so checking the signs can find the overflow. Similarly, we add two negative numbers: -105 + -2= -107. The N-R will represent a number in positive region, (-107↔103). So the sign checking method can be used for complement representation.

Sign determination in RNS is a time-consuming operation. For signs checking overflow detection, we not only need to know the signs of operands before calculation, but also need figure out the sign of result after consistency checking. So this method isn't a good choice.

2.2.3 Excess –M/2 (M/2 =0) overflow detection

a) Add two positive numbers:

1) 1+104 = 105, overflow should be detected.

1(106) = (1,1,0,1,7,2)
104(209) = (2,4,1,6,0,1)
105 = 1+104 = (0,0,1,0,7,3) =>Add Correct factor: (0,0,0,0,1,2)
(0,0,0,0) <=> 0 $|0|_{11} = 0$ $|0|_{13} = 0$ Consistency checking fail
$(|\Delta|_{m5}, |\Delta|_{m6}) = (10,11)$

2) 2+104 = 106, overflow should be detected.

2(107) = (2,2,1,2,8,3)
104(209) = (2,4,1,6,0,1)
106 = 2+104 = (1,1,0,1,8,4) => Add Correct factor: (1,1,1,1,2,3)
(1,1,1,1) <=> 1 $|1|_{11} = 1$ $|1|_{13} = 0$ Consistency checking fail

$(|\Delta|_{m5}, |\Delta|_{m6}) = (10,11)$

Similarly, try all the possible combinations of positive number**, the pair $(|\Delta|_{m5}, |\Delta|_{m6})$ get a fixed value (10,11)**.

b) Add two negative numbers:
1) (-1)+(-105) = -106 , overflow should be detected.
-1(104) = (2,4,0,6,5,0)
-105(0) = (0,0,0,0,0,0)
-106 = (-1)+(-105) = (2,4,0,6,5,0) => Add Correct factor: (2,4,1,6,10,12)
(2,4,1,6) <=> 209 $|209|_{11} = 0$ $|209|_{13} = 1$ Consistency checking fail
$(|\Delta|_{m5}, |\Delta|_{m6}) = (1,2)$
2) (-3)+(-104) = -107 , overflow should be detected.
-3(102) = (0,2,0,4,3,11)
-104(1) = (1,1,1,1,1,1)
-107 = (-3)+(-104) = (1,3,1,5,4,12) => Add Correct factor: (1,3,0,5,9,11)
(1,3,0,5) <=> 208 $|208|_{11} = 10$ $|208|_{13} = 0$ Consistency checking fail
$(|\Delta|_{m5}, |\Delta|_{m6}) = (1,2)$

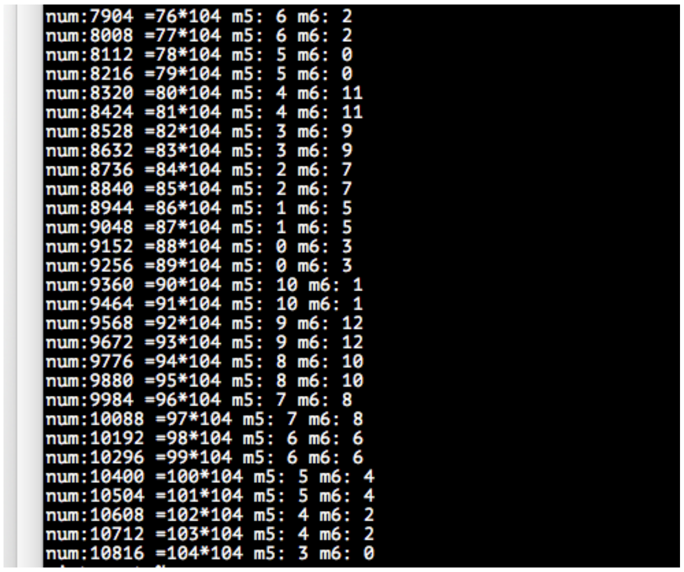
Similarly, try all the possible combinations of negative number, **the pair $(|\Delta|_{m5}, |\Delta|_{m6})$ get a fixed value (1,2).**

Check the error correct table, $(|\Delta|_{m5}, |\Delta|_{m6}) = (10,11)$ and $(|\Delta|_{m5}, |\Delta|_{m6}) = (1,2)$ are not legitimate addresses in Table 1. So we are **possible to distinguish error, overflow and underflow in Excess –M/2 signed number representation method**.

We only discussion addition here. Consistency checking is also suitable for subtraction overflow detection.

2.3 Multiplication Overflow Detection?

From the experiment results, the consistency checking is unable to detect multiplication overflow. The $(|\Delta|_{m5}, |\Delta|_{m6})$ pair values always change and are even possible to overlap with the error correct table entries. So unable to tell the difference between error and overflow.



```
num:7904 =76*104 m5: 6 m6: 2
num:8008 =77*104 m5: 6 m6: 2
num:8112 =78*104 m5: 5 m6: 0
num:8216 =79*104 m5: 5 m6: 0
num:8320 =80*104 m5: 4 m6: 11
num:8424 =81*104 m5: 4 m6: 11
num:8528 =82*104 m5: 3 m6: 9
num:8632 =83*104 m5: 3 m6: 9
num:8736 =84*104 m5: 2 m6: 7
num:8840 =85*104 m5: 2 m6: 7
num:8944 =86*104 m5: 1 m6: 5
num:9048 =87*104 m5: 1 m6: 5
num:9152 =88*104 m5: 0 m6: 3
num:9256 =89*104 m5: 0 m6: 3
num:9360 =90*104 m5: 10 m6: 1
num:9464 =91*104 m5: 10 m6: 1
num:9568 =92*104 m5: 9 m6: 12
num:9672 =93*104 m5: 9 m6: 12
num:9776 =94*104 m5: 8 m6: 10
num:9880 =95*104 m5: 8 m6: 10
num:9984 =96*104 m5: 7 m6: 8
num:10088 =97*104 m5: 7 m6: 8
num:10192 =98*104 m5: 6 m6: 6
num:10296 =99*104 m5: 6 m6: 6
num:10400 =100*104 m5: 5 m6: 4
num:10504 =101*104 m5: 5 m6: 4
num:10608 =102*104 m5: 4 m6: 2
num:10712 =103*104 m5: 4 m6: 2
num:10816 =104*104 m5: 3 m6: 0
```

Figure 1 (|Δ|m5 , |Δ|m6) pair values from multiplication overflow detection tests

Actually, parity checking also can't detect the multiplication overflow. Assume we have a RNS system with all moduli odd. E.g. (3,5,7,11). The odd dynamic range M = 3*5*7*11 = 1155. I list some example in the table below. In line 1000*2 and 1000*3, both of them are overflow. But their compute parity bits are different. So even we change all N-R moduli to odd, it won't help us to easily detect multiplication overflow. I have read several papers about RNS overflow, but all of them only discuss Add/Sub operations. **Any ideas?**

	True result	True parity	Overflow	Compute result	Compute parity
1000*1	1000	even	no	1000	even
1000*2	2000	even	yes	2000-1155= 845	odd
1000*3	3000	even	yes	3000-2310 = 690	even

2.4 Overflow Detection in (199,233,194,239,251,509) system

The discussions in 2.1 to 2.3 are based on moduli (3,5,2,7,11,13). We also make similar verification in (199,233,194,239,251,509). But we only choose some sample points, not all the possible operands combination.

From the results, this add/sub overflow detection method also works in this large moduli system for unsigned number and Excess –M/2 signed number. If the operation results are overflow, ($|\Delta|_{m5}$, $|\Delta|_{m6}$) value pair are fixed and equal to (77, 289). If the operation result are underflow, ($|\Delta|_{m5}$, $|\Delta|_{m6}$) value pair are also fixed and equal to (174, 220). Checking the error correction table in Appendix E of Watson's thesis, both (77,289) and (174,220) are not legitimate address. So we can distinguish the errors and overflow for moduli (199,233,194,239,251,509).

3. A New Residue Numbers Comparison Method

Residue numbers comparison is another main function in RIU. Jen-shiun etc. [1] proposed a number comparison method for residue numbers based on parity bits:

THEOREM 1. *Let X and Y have the same parity and Z = X - Y. X > Y, iff Z is an even number. X < Y, iff Z is an odd number.*

THEOREM 2. *Let X and Y have different parities and Z = X - Y. X >_Y, iff Z is an odd number. X < Y, iff Z is an* even *number.*

But unfortunately, the prerequisites of this parity comparison method are all moduli are supposed to be odd and pair-wise relatively prime. In our design, one of the non-redundant moduli is even, so this trick isn't suitable here. Essentially the parity comparison algorithm is just a subtraction and then checking the overflow. Because the subtract results need consistency checking in RIU, we are possible to use the low cost overflow detection method proposed in Section 2 to generate a new residue number comparison method.

New comparison method: X and Y are two RRNS numbers. X - Y and the result make consistency checking. X>=Y iff $(|\Delta|_{m5}, |\Delta|_{m6})$ is (0,0); X<Y iff $(|\Delta|_{m5}, |\Delta|_{m6})$ is (1,2) for moduli (3,5,2,7,11,13); X<Y iff $(|\Delta|_{m5}, |\Delta|_{m6})$ is (174,220) for moduli (199,233,194,239,251,509).

This new residue number comparison method can be used for both unsigned and Excess –M/2 signed numbers. It's easy to understand that this idea is suitable for unsigned residue numbers. Because if X<Y, the X – Y result will be negative and not in the range [0,M). So the consistency checking will detect the underflow via the value pair of $(|\Delta|_{m5}, |\Delta|_{m6})$. For Excess –M/2 signed numbers X and Y, they have related mapped residue numbers. We call them mapped_X and mapped_Y here. E.g. X =0 => mapped_X = M/2. We can get the conclusion that X>=Y iff mapped_X>= mapped_Y. So the compassion is similar to unsigned number. As we discussed before, the Add/Sub/Mul operations of implicit signed number representation need correction factors. But it should not add correction factor for this comparing subtraction and just treat them as unsigned number, otherwise the compare result may be not correct.

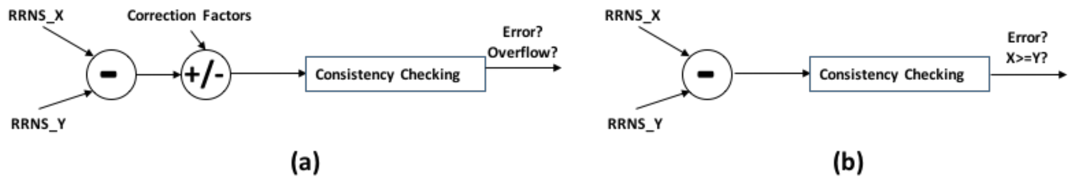


Figure 2 (a) Normal Signed Numbers Subtraction (b) Signed Numbers Compare Subtraction

We have found two other RNS comparison methods from Watson:
1) Make a RRNS subtraction(X-Y). Converting the RRNS result to binary (parallel with consistency checking). If binary result>0, then X>Y. But except the comparison, it seems that we don't need the RRNS to binary converting logic anymore.
2) Relative magnitude comparison algorithm. By now we are not sure whether it's suitable

or could be extended for implicit signed number representation method. Moreover, this need more extra hardware overhead.

This new comparison method may give us some chances to make RIU simple.

4. Comparison about signed number representation methods

Methods	Need Sign determination for correct factor?	Can detect overflow during consistent checking?	Can use the new residue numbers comparison method?
Complement	yes	no	no
Excess –M/2	no	yes	yes

Based on these results, we prefer to use Excess –M/2 method to represent signed numbers.

Reference:

[1] J.-S. Chiang and M. Lu, "Floating-point numbers in residue number systems, " Computers and Mathematics with Applications, vol. 22, no. 10, pp. 127-140, 1991.

Representing real numbers in a Redundant Residual Number System (RRNS) based computer

Sriseshan Srikanth, Bobin Deng, Eric R. Hein, Thomas M. Conte, Erik Debenedictis and Jeanine Cook

Abstract—With the end of Dennard Scaling, computer architects have been looking for novel approaches to improve efficiency. One promising direction involves being able to correct intermittent errors efficiently at runtime due to lowered signal integrity and/or marginal post-CMOS devices. Recent work has proposed such an error correcting computer using the RRNS framework, however, it is limited to integer data. In this paper, the authors propose a mechanism to represent real numbers using the RRNS framework, thereby allowing reliable operation while maintaining reasonable range, precision and performance when compared to those of unreliable binary real numbers.

Index Terms—

1 Introduction

Computers based on the Residual Number System (RNS) have been proposed in the past, especially in the domain of high performance, low energy digital signal processors [5], [6], [10] for the inherent parallelism and low bit width ops that RNS provides. Decades after a redundant component was augmented [11] into RNS, known as Redundant Residual Number System (RRNS), recent work [3] has emerged to design a general purpose compute core complete with ISA and microarchitecture. The underlying principle of such an RRNS computer is that of being able to tolerate intermittent errors not only in storage, but also in compute.

The utility of RRNS error correcting computers stems from the fact that transistor scaling - that allowed smaller transistors to operate at the same power density - has ended [2]. In the absence of such conventional scaling, lowering supply voltage V_{dd} below conservative limits and/or using post-CMOS devices [1] are seen as approaches towards continued increase in performance per Watt. However, such techniques tend to introduce intermittent errors.

RRNS provides a representation that offers error correction (especially for compute) at a lower area and energy overhead than when using the conventional combination of binary representation and Triple Modular Redundancy (TMR) [9]. However, prior work falls short of providing a solution that protects non-integer numbers in RRNS.

This paper proposes an extension to the RRNS representation to allow for reliable storage and arithmetic on real numbers.

2 Background

2.1 Residual Number System (RNS)

The Residual Number System [7] has been used as an alternative to the binary number system chiefly to speed up computation. This increased efficiency comes from the fact that a large integer can be represented using a set of smaller integers, with arithmetic operations permissible on the set in parallel. We present some of the properties of an RNS system without proof.

Let $B = \{b_i \in \mathbb{N}\ for\ i = 1, 2, 3, ..., n\}$ be a set of n co-prime natural numbers, which we shall refer to as bases or moduli. $M = \prod_{i=1}^{n} b_i$ defines the range of natural numbers that can be injectively represented by an RNS system that is defined by the set of bases B. Specifically, for x such that $x \in \mathbb{N},\ x < M$, then, $x \equiv (|x|_{b_1}, |x|_{b_2}, |x|_{b_3}, ..., |x|_{b_n})$, where $|x|_m = x mod\ m$. Each term in this n-tuple is referred to as a residue.

We also note that addition, subtraction and multiplication are closed under RNS. This is because of the following observation: given $x, y \in \mathbb{N},\ x, y < M$, we have $|x\ op\ y|_m = ||x|_m\ op\ |y|_m|_m$, where op is any add/subtract/multiply operation.

2.2 Redundant Residual Number System (RRNS)

To augment RNS with reliability, r redundant bases are introduced. The set of moduli now contains n non-redundant and r redundant moduli: $B = \{b_i \in \mathbb{N}\ for\ i = 1, 2, 3, ..., n, n+1, ..., n+r\}$. The reason these extra bases are redundant is because any natural number smaller than $M\ (= \prod_{i=1}^{n} b_i)$ can still be represented uniquely by its n non-redundant residues. Intuitively, the r redundant residues form a sort of an *error code* because of the fact that all residues are transformed in an identical manner under arithmetic operations. For x such that $x \in \mathbb{N},\ x < M$, then, $x \equiv (|x|_{b_1}, |x|_{b_2}, |x|_{b_3}, ..., |x|_{b_n}, |x|_{b_{n+1}}, ..., |x|_{b_r})$ contains n non-redundant residues as well as r redundant residues.

Upon applying arithmetic transformations to an RRNS number, any error that occurs in one of the residues is contained within that residue and does not propagate to other residues. When required, such an error can be corrected with the help of the remaining residues. Specifically, an RRNS system with (n, r) = (4, 2), a single errant residue can be corrected, or, two errant residues can be detected. Table 1 provides a simple example that illustrates such error correction. Research by Watson and Hastings [8], [11], [12] lays the foundation for the underlying theoretical framework that is used and extended in our work. We use (199, 233, 194, 239, 251, 509) as our (4, 2)-RRNS system, providing a range $M = 199 \times 233 \times 194 \times 239 \in (2^{31}, 2^{32})$.

It can be seen that not only does a residue number system achieve a higher efficiency due to enhanced bit-level parallelism

S. Srikanth, B.Deng, E. Hein and T. Conte are with Georgia Institute of Technology, Atlanta, GA, 30332 USA e-mail: seshan@gatech.edu
E. Debenedictis and J. Cook are with Sandia National Laboratories...
Manuscript received ...; revised ....

Table 1: A (4, 2)-RRNS example with the simplified base set (3, 5, 2, 7, 11, 13). Range is 210, with 11 and 13 being the redundant bases.

Decimal	mod 3	mod 5	mod 2	mod 7	mod 11	mod 13
13	1	3	1	6	2	0
14	2	4	0	0	3	1
13+14=27	(1+2)mod 3=0	2	1	6	5	1
	All columns function independently of one another.					
	An error in any one of these columns (residues) can be corrected by the remaining columns.					

(also, no carries required for addition), but also that introducing 50% of overhead is sufficient to provide resiliency (as opposed to needing a 200% overhead for TMR). As the granularity of an error is that of an entire residue, RRNS is capable of potentially correcting multi-bit errors as well, for *free*.

The remainder of this paper describes real number representation in such an RRNS framework.

3 Proposition

A reliable real number representation system desires the following properties:

1) *Range*. Range that is a superset of the 32 bit IEEE 754 representation (hereafter referred to as *float*). This includes special values: subnormal, ± 0, $\pm\infty$ and *NaN*.
2) *Precision*. Precision that is comparable to that of *float*.
3) *Redundancy*. Amenable to RRNS error check (compute or storage).
4) *Fast*. Minimal arithmetic complexity for performance and energy efficiency.

The proposed representation is designed with the above goals in mind.

Goals *Range* and *Precision* can be naively realized using the same representation as that of *float*. However, floating point arithmetic procedures involve dividing the mantissa by the radix during the normalization processes. For *float*, this radix is 2, and dividing by 2 is a trivial bit shift. However, for *Redundancy*, we require the mantissa to be represented in RRNS form, in which case, such down scaling (divide by 2) is generally expensive. The only form of 'division' that is efficient in RRNS is when the dividend is one of the moduli themselves [11]. **This requires changing the radix to one of the RRNS moduli, which is typically greater than 2.**

Intuitively, we see a trade-off between the goals *Fast* and *Precision*, given that we must achieve *Redundancy*. Before quantifying the impact of changing the radix, we observe Lemma 1.

Observation 1. *Given two radices $r_1, r_2 \in \mathbb{N}$, $2 \leq r_1 < r_2$, for every $x \in \mathbb{R}$ that can be represented as $m_1 r_1^{e_1}$, where $m_1 \in \mathbb{R}$, $e_1 \in \mathbb{Z}$, $1 \leq m_1 < r_1$, x can also be represented as $m_2 r_2^{e_2}$, where $m_2 \in \mathbb{R}$, $e_2 \in \mathbb{Z}$, $1 \leq m_2 < r_2$.*

[m, r, e are mantissa, radix and exponent respectively, in scientific notation parlance.]

Lemma 1. *Assume definitions in Observation 1. Then, m_2 and e_2 can be computed by the following procedure:*

$$\begin{aligned} e_2 &= \begin{cases} \lceil e_1\alpha \rceil & if\ |x r_2^{-\lceil e_1\alpha\rceil}| \geq 1 \\ \lceil e_1\alpha \rceil - 1 & if\ |x r_2^{-\lceil e_1\alpha\rceil}| < 1 \end{cases} \\ m_2 &= x r_2^{-e_2} \\ where,\ \alpha &= log_{r_2} r_1 \\ and,\ \lceil f \rceil &= ceiling(f) \end{aligned}$$

Table 2: Fixed point mantissa representation and dynamic range upon changing radix.

Radix	n_s	n_f	Positive Range
2 (*float*)	0	23	[1.4013E-45, 3.40282E+38]
194	8	23	[6.5057E-296, 7.1097E+290]
239	8	23	[2.5014E-307, 2.2781E+302]
509	9	22	(~0, ~INF)

Proof.

$$\begin{aligned} x &= m_1 r_1^{e_1} = m_2 r_2^{e_2} \\ \implies e_2 &= log_{r_2}\frac{m_1}{m_2} + e_1 log_{r_2} r_1 \\ \because m_1 &\in [1, r_1) \\ and\, m_2 &\in [1, r_2) \\ and\, r_2 &> r_1 \\ \therefore log_{r_2}\frac{m_1}{m_2} &\in (-log_{r_2} r_2, log_{r_2} r_2) \\ \implies |log_{r_2}\frac{m_1}{m_2}| &< 1 \end{aligned}$$

$\because e_2 \in \mathbb{Z}$ and $|log_{r_2}\frac{m_1}{m_2}| < 1$, the proof is complete. □

Note that Observation 1 assumes that m_1 and m_2 can assume any real value in the range [1, r_1) and [1, r_2) respectively. This implies that we have an unlimited supply of bits following a decimal point. In a physical representation, the mantissa must obviously be truncated at the cost of *Precision*.

The *Redundancy* property unique to RRNS is that the 'error codes' are preserved under arithmetic operations. For floating point, these operations operate on the mantissa and the exponent in an independent manner. It is therefore logical to separate out the representation/encoding of the mantissa from the exponent to avoid unnecessary (complex and expensive) extraction from a combined format.

The mantissa m is typically represented as a fixed point number $m^s.m^f$, where m^s and m^f have n_s and n_f bits and represent the significant and fractional portions of the mantissa respectively. In *float*, m^s is either 0 (subnormal) or 1 (recall from Observation 1 that $1 \leq m < radix$); which implies that m^s is implicit and therefore n_s is 0 for *float*.

For our reference RRNS system, the representable range is 31 bits and since the mantissa is encoded distinctly from the exponent, we have $n_s + n_f = 31$. Recall that a requirement to achieve the goal *Fast* is that the radix must be one of the moduli of the RRNS system. As the moduli in our reference system are greater than 2, we observe an increased dynamic *Range* when compared to *float*. Table 2 quantifies these for a few moduli. (Note, the minimum values are obtained using the subnormal representation.)

In order to understand the impact that changing the radix has on *Precision*, we first present a few established notions [15] of precision itself.

3.0.0.1 **Maximum Absolute Representation Error**: MARE is defined as half the distance between two consecutive represented numbers for a given exponent e. For clarity, we denote this distance by the term *gap*, which is equal to $ulp \times r^e$, where ulp, r and e are unit of least precision, radix and exponent respectively.

3.0.0.2 **Maximum Relative Representation Error**: As MARE is not constant across the entire range of representable numbers, MRRE is defined to normalize out the varying exponent term, and is equal to $\frac{1}{2}ulp \times r$.

3.0.0.3 **Average Relative Representation Error**: MRRE is typically considered a good measure if operands are uniformly distributed across the entire range. Upon applying a heuristic density function for the mantissa and computing an average, ARRE is defined as equal to $\frac{r-1}{ln\ r} \times \frac{ulp}{4}$.

A straightforward, intuitive observation from these three figures of merit is that decreasing the *ulp* decreases error, thereby rightly indicating increased precision. However, *ulp* is dictated solely by the number of bits in the fractional part of the mantissa, i.e., n_f. For our reference system, this is rather fixed (*cf.* Table 2).

Having a more or less constant *ulp* leaves us with examining the impact of changing *r* alone. If we must take into consideration the entire range representable, it can be argued that a higher range would result in a lower precision. Imagine the following analogy: the *clarity* (~$precision$) of a picture would deteriorate if the screen size (~r) is increased without changing the screen resolution (~ulp). This is corroborated by the equations for MRRE and ARRE, where the error grows with r and $\frac{r-1}{ln\ r}$ respectively. It is therefore wise to select an r that is as small as possible. For the smallest modulus in our reference system, 194, the MRRE and ARRE increase by a factor of 97 and 25 respectively, when compared to *float*.

However, it is unfair to expect to maintain the precision of *float* across a much higher range, without increasing n_f. MRRE as well as ARRE are good figures of merit only when we are interested in representing the entirety of the increased representable range accurately. For an apples-to-apples comparison with *float*, we restrict our forthcoming *Precision* analysis to the representable range of *float*.

Therefore, for a more thorough understanding of the impact increasing the radix (to 194) has on *Precision*, we examine *gap* (*cf.* paragraph 3.0.0.1 above) at each exponent that is representable by *float*. In Figure 1, for each such exponent x on the X-axis, Lemma 1 is used to compute the exponent $x^{'}$ that would result upon increasing the radix from 2 (*float*) to 194 (smallest RRNS modulus in reference system). The *gap* is computed for each $x^{'}$ (with radix 194) and each x (with radix 2), and their ratio is plotted on the Y-axis. A ratio smaller than 1 indicates that increasing the radix results in higher precision, and a ratio larger than 1 indicates that increasing the radix diminishes precision. We observe that the loss in precision is at most off by a factor of 200. In comparison, the loss in precision due to transitioning from *double* to *float* is of the order of 10^8 for the same (*float*) range.

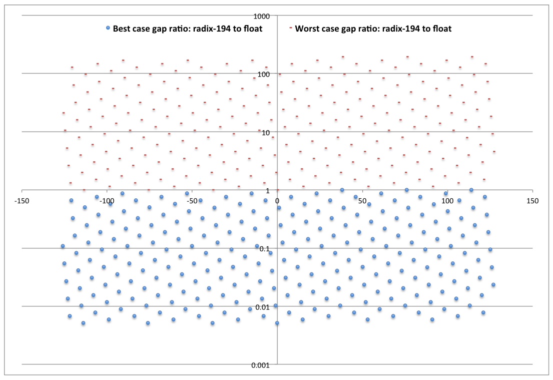


Figure 1: Ratio of *gap* (*cf.* paragraph 3.0.0.1) with radix 194 to that with radix 2 (*float*) over a range of exponents spanning that of *float*. A lower *gap* indicates higher precision. In comparison, the *gap* ratio of *float* to *double* is of the order of 10^8.

We conclude that enforcing the goals *Redundancy* and *Fast* with a budget constrained by the reference RRNS system results in a *Precision* that is comparable to that of *float*, over a similar *Range*.

The representation of sign for mantissa and exponent depends on two factors: (1) Floating point arithmetic requires explicit prior knowledge of the signs of the operands for efficient operation, (2) The CREEPY representation must represent special values that are also represented by *float*: subnormal, ± 0, $\pm\infty$ and *NaN*. The upshot of these two factors is that the mantissa is in explicit sign-magnitude format (where the sign bit and magnitude bits have distinct representations) and that the exponent is in implicit signed format.

Implications on Microarchitecture

We now discuss further details of the representation, and its impact on the microarchitecture of an RRNS computer such as [3].

Recall that operations pertaining to each RRNS modulus are handled by a subcore [3]. For the reference RRNS system considered in this paper, there are 6 subcores, one for each modulus. The sign bit is required by each RRNS subcore, and hence must be replicated across them. Similar to a control signal, TMR is employed for the sign bit and is placed within the register file, with the check being done prior to being read and replicated by the subcores. An incorrect sign bit is analogous to an incorrect opcode as far as floating point arithmetic is concerned.

Recall that the mantissa is represented distinctly from the exponent; this now implies that the magnitude of the mantissa has the entire range of RNS numbers to itself. Therefore, 49 RRNS bits are used to represent the magnitude of the mantissa. Recall from Table 2, the mantissa is in a fixed point form, with the point location implicit.

Finally, the exponent also has the entire range of RNS numbers (32bits) to itself, however, given the range/precision analysis and the representability of special values, it is more than sufficient to represent 8 bits' worth of non-redundant exponent, *i.e.,* same as *float*. There are two options to protect these 8 bits:

1) A smaller RRNS system whose range is of the order of 2^8 (rather than 2^{31}). The implicit signed representation would then be of the form of *excess-* $\frac{M^{'}}{2}$, where $M^{'}$ is the range of the smaller RRNS system. However, this

requires an additional set of RRNS check hardware as well as an additional set of subcores/adders that are designed for the smaller RRNS system.

2) Perform replication and TMR in a manner identical to that of the mantissa sign bit.

In summary, an RRNS floating point register would logically contain 3 (mantissa sign, TMR) + 24 (exponent, assuming TMR) + 49 (mantissa magnitude in RRNS) = 76 bits. A smaller storage footprint can be achieved by employing a smaller RRNS system to represent the exponent, as discussed above. Further microarchitectural optimizations and more compact representations are plausible, but are out of scope of this paper.

Modification to Floating Point arithmetic

In this subsection, we highlight changes to the fundamental algorithms for addition and multiplication.

Addition

The baseline algorithm for addition of two real numbers involves three stages:

1) *Denormalize*. Scale the mantissa of one of the numbers such that exponents of both numbers match.
2) *Add*. Add the two mantissas.
3) *Renormalize*. Scale the exponent of the result such that the mantissa is of normalized form, as noted by Observation 1.

With the proposed RRNS real number system, we note two changes:

- *Denormalize* / *Renormalize*: Scaling is now done by 194 (the smallest RRNS modulus) rather than by 2. While up-scaling is a simple matter of RRNS multiplication, down-scaling requires a special algorithm, the cost of which is 3 subtractions, 2 multiplications and 3 table lookups [11].
- *Add*. Regular RRNS addition can be employed.

Multiplication

The baseline algorithm for multiplying two real numbers is less involved than addition. The mantissas can be multiplied, the exponents can be added and a renormalization can be done on the final result.

Upon shifting to RRNS, we note the following:

- Exponent addition is rather trivial, the exact algorithm depends upon the representation of the exponent in the microarchitecture (*cf.* Section 3).
- Renormalization is as discussed for the case of addition above.
- Mantissa multiplication is that of two fixed point numbers, meaning that additional intermediate bits are necessary to preserve precision. However, with RRNS, this is non-trivial. Therefore, we suggest first scaling the mantissas to fractions, and then performing fractional multiplication [11] before scaling back.

It can be seen that function units performing arithmetic operations on RRNS real numbers have a logically increased (sub)pipeline depth, meaning that high performance scientific applications could benefit from increased throughput, albeit at an increased pipeline fill latency.

4 Related Work and Conclusion

Prior work in the Digital Signal Processors (DSP) domain [5], [6], [10] had neither the redundant aspect of RRNS, nor did they have the need to handle real numbers (fixed point representation is deemed sufficient for DSPs).

There have been proposals to represent floating point numbers in RNS [4], [13], [14]. However, these place restrictions on the RNS bases rendering them in-extensible for purposes of RRNS. The closest work that achieves reliable real number handling is the classic RRNS paper [8], [11], [12]. However, as their work predates IEEE *float*, their solution lacks in completeness from the point of view of today's scientific community. It must be noted, however, that our proposition for real RRNS numbers leverages their error correction, down scaling by RNS modulus, and fractional multiplication algorithms.

In this paper, we have proposed a real RRNS number representation and associated algorithms/microarchitecture extensions to an integer RRNS computer [3] that augment reliability while remaining comparable to *float* w.r.t. range, precision and performance.

References

[1] D. E. Nikonov and I. A. Young, ”Overview of Beyond-CMOS Devices and a Uniform Methodology for Their Benchmarking,” in Proceedings of the IEEE, vol. 101, no. 12, pp. 2498-2533, Dec. 2013.
[2] M. Adrian, *The end of Dennard scaling*, 2013.
[3] B. Deng, S. Srikanth, E. R. Hein, P. G. Rabbat, T. M. Conte, E. DeBenedictis, J. Cook, *Computationally-Redundant Energy-Efficient Processing for Yall (CREEPY)*, to appear in the IEEE International Conference on Rebooting Computing 2016.
[4] J. S. Chiang, M. Lu, *Floating-point numbers in residue number systems*, Computers and Mathematics with Applications,22, no. 10, 127-140, 1991.
[5] R. Chokshi , K. S. Berezowski , A. Shrivastava , S. J. Piestrak, *Exploiting residue number system for power-efficient digital signal processing in embedded processors*, Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems, October 11-16, 2009, Grenoble, France
[6] E. D. Claudio, F. Piazza and G. Orlandi, *Fast Combinatorial RNS Processors for DSP Applications*, IEEE Trans. Computers, vol. 44, no. 5, pp. 624-633, May 1995.
[7] H. L. Garner, *The Residue Number System*, IRE Transactions on Electronic Computers, pp. 140-147, June 1959.
[8] C. W. Hastings, *Automatic detection and correction of errors in digital computers using residue arithmetic*, in Proc. 1966 IEEE Region Six Annu. Conf, pp. 429-464.
[9] R. E. Lyons, W. Vanderkulk. *The use of triple-modular redundancy to improve computer reliability*, IBM Journal of Research and Development 6.2 (1962): 200-209.
[10] J. Ramirez, *RNS-enabled digital signal processor design*, Electron. Lett., vol. 38, no. 6, pp. 266-268, 2002
[11] R. W. Watson, *Error detection and correction and other residue interacting operations in a residue redundant number system*, Univ. California, Berkeley, 1965.
[12] R. W. Watson and C. W. Hastings, *Self-checked computation using residue arithmetic*, Proc. IEEE, vol. 54, pp. 1920-1931, Dec. 1966.
[13] J. O. Tuazon, *Residue number system in computer arithmetic*, Iowa State university, 1969.
[14] R. Dhanabal, V. Barathi, S. K. Sahoo, N. R. Samhitha, N. A. Cherian, P. M. Jacob, *Implementation of floating point MAC using residue number system*, Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on. IEEE, 2014.
[15] I. Koren. *Computer arithmetic algorithms*, Universities Press, 2002.

DISTRIBUTION:

1	MS 1082	Patrick Chu, 05265
1	MS 0359	Greg C. Frye-Mason, 01171
1	MS 1318	Robert J. Hoekstra, 1422
1	MS 1322	John B. Aidun, 1425
1	MS 0359	D. Chavez, LDRD Office, 1911
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories